Universitetet i Oslo
Institutt for informatikk

Vurdering av

generiske typer

i programmeringsspråket

- JAVA -

Hovedfagsoppgave av

Endre Meckelborg Rognerud

Februar 2001

Forord

Denne rapporten er resultatet av min jobbing med hovedfag på Institutt for informatikk (Ifi) på Universitetet i Oslo (UiO) i perioden 1999-2001.

Arbeidet med denne oppgaven har vært lærerikt og ikke minst utfordrende. Jeg, som de fleste hovedfagsstudenter, har "smertelig" fått erfare hvor tungt det er å ha en stor oppgave hengende over en i lang tid. Selv i mitt "frisemester", hvor jeg skulle koble av studiet med jobbing på Amatøren, puben på Sogn studentby, gikk jeg stadig med dårlig samvittighet.

Likevel er jeg glad jeg har fått med meg erfaringen som oppgaven har gitt. Det har vært lærerikt både med tanke på og sette seg dypt inn i et problemområdet, samtidig som det er første gang jeg har skrevet noe så omfattende.

... and I wish to thank ...

Først og fremst må jeg takke min veileder, Stein Krogdahl, for hans store engasjement og pågangsmot. I perioder har dette vært absolutt nødvendig når jeg midt i hovedfagsdepresjonen, ikke akkurat hadde den gløden som jeg burde hatt :)

Jeg må også takke Bjørn Willassen, Elin Bardal og Birger Møller-Pedersen for nyttige kollokvier i starten av hovedfagsstudiet.

I tillegg vil jeg takke Universitetets senter for informasjonsteknologi (USIT), hvor jeg har jobbet ved siden av studiet. Det er langt mellom arbeidsplasser som har like spennende arbeidsoppgaven og en like fleksibel arbeidstid som det jeg hadde på USIT.

Endre Meckelborg Rognerud

Innholdsfortegnelse

Figurliste

[*] Figuren er hentet fra boka *"Dilberts lov om ledelse"* skrevet av Scott Adams. Management Fads & Other Workplace Afflictions, 1996. Norsk utgave utgitt av Bladkompaniet AS, 1998.

1 Innledning

En viktig faktor for å høyne produktiviteten og kvaliteten i produksjon av programvare, er det å kunne operere med separat programmerte og vel avgrensede ”programbiter”, som i størst mulig grad er ferdig testet ut og som tilbyr et veldefinert grensesnitt. Disse må også på en fleksibel måte kunne hentes inn i programmer som har behov for den funksjonaliteten som programbitene tilbyr.

Jeg vil helt generelt bruke benevnelsen ”pakker” på slike programbiter, og de vil typisk være en del av større biblioteker som enten leveres sammen med språksystemer, eller som tilbys på et mer eller mindre kommersielt marked. Når jeg etter hvert vil komme inn på objektorienterte språk, som jeg i hovedsak skal vurdere, vil pakkebegrepet ofte erstattes med klasser eller samlinger av klasser.

En viktig ting ved slike pakker er at de må ha en viss generalitet, slik at en enkelt pakke kan hentes inn og fungere hensiktsmessig i flest mulig programmer som på en eller annen måte har behov for pakkas funksjonalitet.

Ønsket om generalitet kan gjelde mange ting, men om man arbeider i tradisjonelle, statisk typede språk, er det ett problem man stadig kommer borti; nemlig at typene som er brukt i en pakke, meget nær må stemme med de typene som brukes i det programmet pakkene skal hentes inn i. Ellers kan det være vanskelig eller umulig å få de forskjellige pakkene til å fungere sammen. Om det ikke finnes spesielle mekanismer som gjør det enklere for en bruker å programmere med generelle typer, vil det derfor være vanskelig å lage pakkene slik at de kan benyttes i forskjellige kontekster.

Et aller enkleste eksempel her kan være at man ønsker å skrive en sorteringsrutine som skal virke for både heltall, reelle tall og komplekse tall. I et statisk typet språk, uten spesielle mekanismer for å takle dette problemet, kan dette være mer eller mindre umulig.

En nærliggende måte å løse denne typeproblematikken på, er å la de typene som vil variere med de forskjellige kontekstene som pakkene skal brukes i, være parametere til pakkene. Når en pakke så skal brukes, kan man så angi de aktuelle typene som gjør at pakka passer inn i det aktuelle programmet. En slik mekanisme kalles gjerne ”generiske parametere”, ”generiske typer”, ”generiske pakker” eller lignende.

Det kan virke helt kurant å lage en slik mekanisme, men avhengig av detaljene i det aktuelle språket og dets typesystem, er det mange ting som må på

plass. Mange språk har imidlertid definert slike generiske mekanismer, og to kjente eksempler er generiske pakker i Ada og templates-mekanismen i C++.

I Java er det, i den nåværende standard, ikke definert noen slik mekanisme. Dette kan synes merkelig, da så mye av språkets funksjonalitet helt fra starten var ment å ligge i de bibliotekene som følger med Java-systemet. Etter hvert som språket har fått større utbredelse og det har blitt tatt i bruk i mange sammenhenger, har behovet for slike mekanismer imidlertid meldt seg tydeligere.

Sun Microsystems, som har definert og utviklet standarden av Java, har også nå annonsert at de vurderer å innføre en slik mekanisme i språket. Det pågår derfor en debatt om hvordan en slik generisk mekanisme bør utformes. Blant annet har det blitt presentert flere artikler rundt temaet i forbindelse med konferansene ECOOP[1] og OOPSLA[2].

1.1 Fokus i oppgaven

Denne hovedoppgaven vil ta for seg noen av de forslagene som er framsatt i denne debatten og gjøre en vurdering av dem blant annet ved å forsøke å bruke mekanismene i omgivelser der kravene til generisk fleksibilitet er spesielt høye.

I de forskjellige artiklene som presenterer forslag til mekanismer, er det enkelte eksempler som går igjen og igjen i forbindelse med vurdering av ”styrken” til forskjellige mekanismer. Jeg har i min oppgave plukket ut noen av disse og bruker dem under min evaluering.

I tillegg har jeg vurdert de forskjellige mekanismene opp mot litt mer generelle eksempler, og som et utgangspunkt for å lage slike omgivelser, har jeg valgt å se på såkalte *”design-patterns”*. Dette er generelle bruksmønstre, gjerne uttrykt ved hjelp av klasser og objekter, men som ikke uten videre lar seg implementere i ferdig skrevne bibliotekklasser.

De er ofte er så abstrakte og ”distribuerte”, at det vil kreve spesielt mye av den generiske mekanismen, for at det skal være mulig å implementere et pattern som en bibliotekspakke. I hvert fall på en slik måte at det enkelt vil være mulig å hente det inn i et program og få det til å fungere der som om det var spesialskrevet for konteksten. Enda vanskeligere vil det være å hente inn flere forskjellige design-pattern i samme program, og få disse til å spille sammen.

I tillegg er programmering med ferdige pakker, moduler etc. noe som er i vinden for tiden. Man kan finne ”hyllemeter” med bøker rundt disse temaene.

---

[1] ECOOP er en årlig europeisk konferanse rundt objektorientering, med navn *”European Conference for Object-Oriented Programming”*. Hjemmeside: http://www.ecoop.org/.

[2] OOPSLA er en tilsvarende konferanse som ECOOP, men hvor konferansen foregår i USA. Forkortelsen står for *”Object-Oriented Programming, System, and Applications”*. Hjemmeside: http://oopsla.acm.org/.

Det hele bygger oppunder ønsket om å programmere så ”rent” og feilfritt som mulig, hvor det er stort fokus på vedlikehold og gjenbruk av kode.

I boka *”Essential Java Style – Patterns for Implementation”* [1] skriver forfatteren om bruk av kjente patterns og andre begreper som kan gjøre kode mer lettlest og vedlikeholdbar. Forfatteren understreker viktigheten av dette.

> *”Code almost never exists in a vacuum – someone will be reading or maintaining your code in the future. [...] If you respect your fellow developer, you will strive for clear code.”*

På en måte kan man se for seg at man skal kunne bygge et program ved i størst mulig grad å sette sammen vel avgrensede og ferdig uttestede pakker. I utgangspunktet kan man si at det er to måter å sette biter sammen til programmer på:

1. **La prosesser samarbeide**. Man ”kjøper” ferdige prosesser, og lar dem samarbeide gjennom et eller annet meldingsbasert system med kjent grensesnitt. Dette kan gjøres ved hjelp av RPC, CORBA osv.
2. **Benytte bibliotekspakker**. Man kjøper ferdige bibliotekspakker som kan linkes inn sammen med ”hovedprogrammet”, og danner til sammen et program som går som én prosess.

Det er mye snakk om den første varianten for tiden, blant annet fordi det også gir grunnlag for distribuerte systemer. I denne hovedoppgaven skal jeg imidlertid først og fremst se på sammensetninger av type 2.

En viktig problemstilling i oppgaven vil derfor være om de foreslåtte generiske mekanismene gjør en slik implementasjonen enklere, og i hvilken grad de forskjellige forslagene fungerer i ulike sammenhenger.

I tillegg til forslagene som er kommet opp i debatten omkring generiske mekanismer i Java, tar også oppgaven for seg noen tilsvarende forslag som tidligere er framsatt for Simula. For disse må man først vurdere hvordan de best kan utformes i ett Java-lignende språk, og deretter vurdere om de kan konkurrere i en eller flere brukssammenhenger med noen av de forslagene som er nevnt over.

En god del forhold vil være viktige når disse vurderingene skal gjøres. I neste delkapittel går jeg, for oversiktens skyld, raskt gjennom noen av de viktigste av disse, og de fleste av dem vil jeg komme fyldigere tilbake til senere i oppgaven.

1.2 Kort om generiske mekanismer

Utviklingen av programmeringsspråk har pågått i mesteparten av datamaskinens historie, og etter hvert har det oppstått et ønske om mekanismer som gjør det enklere med generisk programmering. Innføringen av objektorienterte begreper kan også, fra en passelig synsvinkel, sees som et forsøk på å innføre en delvis løsning på typeproblematikken man har i statisk typede språk.

Ved hjelp av pekere til klasser og subklasser, har man i utgangspunktet mer fleksibilitet i typesystemet. Man kan type variable med en viss klasse, og benytte ulike subklasser av denne i forskjellige kontekster. Slik kan man få til *en del* av fleksibiliteten som det generiske mekanismer gir. Bruk av super- og subklasser kan blant annet løse problematikken man har i forbindelse med sorteringsalgoritmen jeg nevnte tidligere i kapittelet.

Denne tankegangen har flere språk, inkludert Java og Eiffel, utnyttet til gangs. I disse språkene har alle klasser automatisk én felles ytterste superklasse, og i Java heter denne klassen `Object`. Dermed kan man benytte den felles superklassen alle steder hvor man ønsker en generell type. I Javakretser refereres en slik programmering ofte til å programmere ved hjelp av *det generiske idiom*.

Denne metoden tilbyr altså til en viss grad generisk fleksibilitet, og det er ved hjelp av dette at Java tross alt kan leveres med et så stort og fleksibelt klassebibliotek som det Java gjør; selv uten noen eksplisitt generisk mekanisme inkludert i språket.

Men som jeg tar opp i neste kapittel, skaper programmering ved hjelp av det generiske idiom, enkelte problemer som man ønsker å unngå. Blant annet må et program som eksekverer, hele tiden sjekke hvilke typer som faktisk benyttes, og en programmerer må ofte spesifikt angi den konkrete typen til pekere for å få tak i innholdet av et objekt. Disse problemene vil bli beskrevet nærmere i det neste kapittelet.

Ved innføring av en generisk mekanisme i et språk, vil det være en rekke egenskaper ved den valgte mekanismen som det er interessant å karakterisere og vurdere. Noen viktige faktorer er følgende:

- Hvor generelt kan man uttrykke seg i en pakke innen den rammen som den generiske mekanismen gir?
- Hvor fleksibel er den mekanismen som henter en eller flere pakker inn og får det hele til å jobbe sammen som ett program?
- I hvilken grad kan en selvstendig skrevet pakke fullstendig sjekkes av kompilatoren, både syntaktisk og semantisk, og i hvilken grad må noe av denne sjekkingen vente til man ser hvordan pakka brukes?
- For de pakkene som kan sjekkes fullstendig: I hvilken grad kan en kompilator generere ferdig maskin-/bytekode slik at en pakke kan legges inn i et bibliotek og benyttes direkte uten noe videre behandling?

Tidligere i innledningen har jeg hele tiden referert til statisk typede språk, og grunnen til dette er at behovet for en fleksibel generisk mekanisme, vil være større her enn i språk som bygger på andre typefilosofier.

1.3 Statisk og dynamisk typede språk

Forskjellige programmeringsspråk bygger på forskjellige språkfilosofier, og typesystemet er en av forskjellene fra et språk til et annet. Blant annet går det et klart skille mellom statisk og dynamisk typede språk.

I statisk typede språk er typen, i tillegg til å være knyttet opp mot verdien, også knyttet opp mot variabler, metoder og andre typede konstruksjoner i språket. I disse språkene er typene til verdiene og variablene nødt til å stemme overens, ellers vil programmet føre til kompilatorfeil.

Et annet alternativ, som dynamisk typede språk benytter, er å kun la typen være knyttet opp mot en verdi, mens variable og lignende, ikke opererer med typer. Med andre ord kan en variabel holde på verdier av forskjellige typer uten noen problemer. I disse språkene, for eksempel Lisp og Smalltalk, er det ikke på samme måten behov for generiske mekanismer siden man bare opererer med én type variabel.

På mange måter kan en slik filosofi være fordelaktig, men en ulempe er at programeksekveringen går tregere fordi man hele tiden må sjekke om de angitte operasjoner er lovlige for de verdiene som er angitt som operander. Man mister også i stor grad muligheten for å fange programmeringsfeil under kompilering.

1.4 Programmeringsspråket Java

Siden denne oppgaven tar utgangspunkt i programmeringsspråket Java, vil jeg her kort oppsummere de viktigste egenskapene ved dette språket og prøve å plassere det grovt i forhold til andre språk.

Java er utviklet av Sun Microsystems. Det er utviklet for mindre enn 10 år siden, og det har derfor kunnet trekke på erfaringer fra mange andre objektorienterte språk. Noen sier at Java er et slags nytt Simula i C++ sin drakt. I dette ligger det at man har valgt å gå litt tilbake til Simula når det gjelder kjøresystemet (blant annet ved at det har automatisk ”søppeltømming”), samtidig som man i stor grad har kopiert syntaksen til ett av de mest utbredte språkene, C++.

Java leveres med et stort klassebibliotek, og det kan være en stor utfordringen å bli fortrolig med alle klassene og hele klassestrukturen. En viktig side av Java er at det har god støtte for programmering med tråder, nettverk, web og databaser.

Java er ”sikkert” i den forstand at kompilatoren er ganske strikt; i tillegg til at et program automatisk vil få kompilert inn et ganske stort kjøresystem. Det er lagt opp til at mest mulig skal kunne sjekkes av kompilatoren, men det er likevel visse språkkonstruksjoner som må testes under kjøring på samme måte som det blir gjort i Simula, Beta og faktisk også dels i C++. Java er dermed ikke fullt statisk typet.

Første versjon av Java var ferdig i 1995 og det er dermed et ganske nytt språk. Selv om programmerere rundt om i hele verden har begynt å ta språket i bruk, kommer det fremdeles jevnlig nye versjoner av standarden. Noen endringer er større og mer omfattende enn andre, og det er det medfølgende biblioteket som har vært mest utsatt for forandringer.

En av hovedforskjellene mellom Java og de fleste andre programmeringsspråk, er at Java-programmer vanligvis ikke kompileres ned til maskinkode som kjøres direkte, men i stedet kompileres de til en spesiell *"bytekode"* som man må eksekvere via en såkalt Java Virtual Machine (JVM). På den måten klarer Sun å la kompilerte Java-programmer være plattformuavhengige, men det går noe ut over utføringshastigheten.

Dette skaper en del begrensninger når det gjelder utvidelser av språket. På grunn av at folk allerede har tatt i bruk JVM-er over alt i verden, er det vanskelig å gjøre store endringer på bytekoden. Hvis dette skjer, er det mange som ikke får kjørt de nye programmene fordi de sitter på gamle JVM-er. Sun har derfor sagt at utvidelser av Java ikke skal gå lenger enn at gamle JVM-er skal kunne håndtere dem. Eller rettere sagt har de vel uttalt at slike endringer vil føre til at språket får et nytt navn.

Debatten rundt innføring av generiske mekanismer i Java, har ut fra dette blitt noe "snever" siden de fleste har begrenset seg til å foreslå mekanismer som kan implementeres innenfor de rammer som JVM-en har satt. I denne oppgaven vil jeg se mer bort i fra disse restriksjonene, og jeg vil derfor ha friere tøyler på dette punktet.

1.5 Arbeidsmetode og oppbygning av rapporten

Da jeg begynte og jobbe med oppgaven, leste jeg flere artikler som presenterte forslag til mekanismer for generiske typer i Java. Jeg leste også litt om hvordan dette var løst i andre språk, og ut i fra dette, valgte jeg ut fire forslag til generiske typer i Java som jeg skal se nærmere på. Disse vil bli presentert én etter én i kapittel 3. For å ha et grunnlag for en sammenligning og evaluering av de generiske mekanismene, vil jeg i kapittel 2 se på forskjellige kriterier som generiske mekanismer kan bedømmes etter.

I tillegg til disse artiklene, har jeg også sett på en mekanisme som i sin tid ble presentert i forbindelse med Simula. Artikkelen *"A Module-mechanism for Flexible Code Reuse"* [9] skrevet at Krogdahl, Wang, Jensen og Piene, legger fram et forslag til en generisk mekanisme for Simula. Denne har jeg endret på slik at den passer mer til Java, og jeg vil i kapittel 4 se nærmere på hvordan denne mekanismen er i sammenligning med de andre.

I forbindelse med avansert bruk av generiske mekanismer, melder det seg et behov for multippel arv, og jeg vil i slutten av kapittel 4 se litt på begrepet. Multippel arv er ikke implementert i nåværende versjon av Java, men i objektorienterte språk kan det raskt oppstå et behov for dette ved utvikling av

større datastrukturer. Jeg vil i dette kapittelet belyse forskjellige sider av multippel arv, og også vurdere hvordan dette kan implementeres på en god måte.

Den siste delen av oppgaven er å forsøke og bruke de foreslåtte mekanismene i praksis og se i hvilken grad det vil hjelpe en programmerer å benytte generiske mekanismer ved utvikling av generelle klasser og moduler. I denne delen tar jeg utgangspunkt i forskjellige design-patterns som er presentert i boka *"Design Patters – Elements of Reusable Object-Oriented Software"* [2] skrevet av Gamma, Helm, Johnson og Vlissides.

Jeg har plukket ut tre design-patterns som jeg skal se nærmere på, og til slutt har jeg konstruert et større eksempel hvor jeg har kombinert disse tre patternene. Implementasjonen og eksemplene rundt disse står i kapittel 5. Her prøver jeg også, ut i fra mine erfaringer, å trekke noen konklusjoner om styrkene og svakhetene til de framstilte forslagene av generiske mekanismer.

Kapittel 6 er en oppsummering av oppgaven, med noen avsluttende kommentarer.

1.6 Språklige og tekniske valg

Bruk av engelske ord er aldri helt kurant i norske tekster, og IT-faget er spesielt utsatt fordi mange ord og uttrykk som benyttes i det daglige, blir brukt uten at man har funnet en norsk oversettelse. I utgangspunktet mener jeg at vi i Norge er alt for dårlig til å oversette engelske ord, men likevel har jeg i denne rapporten valgt å beholde flere engelske uttrykk selv. Dette har jeg gjort fordi jeg ikke vet om gode norske oversettelser, og bruk av hjemmelagede varianter kan gjøre oppgaven vond å lese.

I starten prøvde jeg for eksempel å benytte ordet *designmønster* framfor *design-patterns*, men etter hvert fant jeg ut at jeg ikke ville oversette uttrykket likevel. Det samme gjelder ordet *casting* (altså eksplisitt å angi at typen til et uttrykk skal forandres), der jeg en stund var inne på oversettelser som *typespesifisering* og *kasting*.

De steder jeg har benyttet engelske ord, fører dette selvfølgelig til at enkelte flertallsformer kan bli litt merkelige, men jeg har funnet ut at jeg likevel likte dette bedre enn å bruke norske varianter som jeg direkte har oversatt selv.

Alle kodeeksempler har jeg implementert i engelsk. Dette har jeg valgt fordi en del egennavn som navn på design-pattern, er engelske, i tillegg til at kodeeksempler jeg har "lånt" fra andre, også har vært implementert i engelsk.

De fleste objektorienterte språk, inkludert Java, støtter forskjellig grad av tilgjengelighet (`public`, `private`, `protected` osv.) på klasser og attributter i klasser. I denne oppgaven hvor jeg har mange utsnitt fra programeksempler, har jeg stort sett valgt å ikke ta med dette, da det kanskje kan forvirre like mye som det er fordelaktig.

Figurene som brukes i rapporten, er stort sett representert ved hjelp av Unified Modeling Language (UML), versjon 1.3. UML er en standard som ser ut

til å slå gjennom, og ved å bruke et standardisert språk, kan man enklere forhindre uklarheter som ellers kunne forekommet. Som oppslagsbok rundt UML, har jeg blant annet benyttet UML-spesifikasjonen [16] utgitt av Object Management Group, Inc (OMG).

Jeg har valgt å ikke lage noe eget kapittel/avsnitt som forklarer UML-elementene jeg benytter, og dette valget har jeg tatt fordi jeg regner med at teksten i sammenheng med figurene vil være nok til at man vil skjønne dem. Dessuten er UML en standard som etter hvert har fått en ganske utbredt bruk.

2 Generiske mekanismer

Som jeg var inne på i innledningen, er det mange problemer som er relatert til design av og programmering med generiske typer. Jeg vil i dette kapittelet gå nærmere inn på noen av disse, og ved hjelp av eksempler, illustrere forskjellige typer problemer som naturlig dukker opp.

Jeg vil først gå litt dypere inn på forskjellige typesystemer, og se på hvor fleksible disse er og på noen problemer som er tilknyttet hvert enkelt av dem. Etterpå viser jeg et eksempel på generisk programmering i et ikke-objektorientert språk, og går deretter inn på hvordan dette spiller seg ut for objektorienterte språk.

Til sist i kapittelet vil jeg ta for meg fire konkrete programmeringssituasjoner. De er av økende ”vanskelighetsgrad”, og jeg vil belyse hva som er vanskeligheten i hvert eksempel. Disse vil jeg benytte videre i kapittel 3 når jeg sammenligner de forskjellige forslagene til generiske typer implementerte i Java.

2.1 Forskjellige typefilosofier og generiske behov

Programmeringsspråkene vi har i dag, har utviklet seg over lang tid. Nye mekanismer og muligheter har blitt lagt til fra språk til språk og fra versjon til versjon. Etter hvert som utviklingen har gått, har språkenes typefilosofier også tatt forskjellige retninger, for eksempel med henhold til statiske kontra dynamiske typesystemer.

Tradisjonelle statisk typede programmeringsspråk, for eksempel Pascal, har et relativt stivt typesystem. Her må alle variabler deklareres, og typene må stort sett bestemmes under kompilering.

Statisk typede språk som ikke er objektorienterte, kan fort få et litt for stivt typesystem, og man vil erfare at man av og til kan ønske seg noe mer fleksibelt. Men i et selvstendig, ”monolittisk” program uten særlig gjenbruk av kode, vil man kanskje ikke merke typesystemets infleksibilitet på samme måte som når man prøver å skrive generelle pakker som er beregnet for biblioteker.

Som jeg nevnte i innledningen, finnes det en helt annen typefilosofi hvor all typing er fullstendig dynamisk. Lisp og Smalltalk er språk som benytter denne form for typing. I slike språk deklarerer man også alle variabler, men man spesifiserer ikke hvilken type de har. Med andre ord kan de senere tilordnes verdier av ulike typer.

Språk med et slikt typesystem har sine fordeler, men samtidig også sine ulemper. Siden typen nå kun er knyttet opp mot verdien i stedet for både mot verdien og variabelen, forsvinner egentlig de fleste av problemene som oppstår i forbindelse med å sette sammen programbiter som er skrevet uavhengig av hverandre. En variabel kan innholde forskjellige typer uten at man trenger å spesifisere det på noen som helst statisk måte.

Internt under programutførelsen er slike systemer likevel nødt til å lagre typen på hvert dataelement, blant annet for å vite hvilke operasjoner som er lovlige på ulike variable. Dette medfører at når man ber om å få utført en operasjon, må systemet først sjekke at operasjonen er lovlig for den aktuelle typen, og deretter finne hvilken variant av operasjonen som gjelder for den aktuelle typen.

Siden disse testene utføres hele tiden, reduseres hastigheten på programeksekveringen. Dette problemet unngås hvis typene bestemmes ved kompilering. I tillegg vil en kompilator for et fullstendig statisk typet språk, ha mulighet til å finne flere feiltyper som ellers ikke ville bli oppdaget før under programeksekvering.

2.2 Språklig integrasjon av mekanismen

En generisk mekanisme lar en programmererer skrive generelle pakker hvor man ikke fullstendig angir typene som skal brukes. Disse må derfor angis senere når pakka benyttes i et annet program. Forskjellige mekanismer opererer med ulike metoder for å ordne dette.

Noen mekanismer bruker en preprosessor som bytter ut de generiske typene før et program kompileres, mens andre lar de generiske typene være en del av selve programmeringsspråket og dets kompilator. Denne forskjellen uttrykker en grad av språklig ”integrasjon” av den generiske mekanismen.

Jeg har prøvd å dele opp mekanismene i tre forskjellige klasser, hvor den generiske mekanismen i økende grad er integrert i kjernespråket:

- Den enkleste formen er en ren makrobasert mekanisme som bygger på at en preprosessor går gjennom programkoden og erstatter de generiske parametere med reelle typer før selve programmet kompileres. Dette er en ren tekstlig metode hvor hver ny ”instansiering” resulterer i ny programkode, og hvor de forskjellige instansieringene dermed ikke står i forbindelse med hverandre. Siden oversettelsen her kun er en tekstlig gjennomgang, kan man også benytte metoden til å bytte ut andre kodeelementer enn bare typer.
- I mellomklassen er de generiske mekanismene en del av språket, og pakkene har dermed en strikt syntaks. De formelle generiske parameterne er imidlertid ikke fullstendig spesifisert, så man kan ikke foreta en full semantisk sjekk av pakkedefinisjonene. På samme måte som for makrobaserte mekanismer, blir dette først gjort når man har satt inn aktuelle parametere på bruksstedet.

- Den siste og mest integrerte varianten, er den som i tillegg til alt i forrige punkt, også har full spesifisering av parametere og full semantisk sjekk av pakkedefinisjonene. En slik oversettelse kan gjøres både ”heterogent” og ”homogent” (se under), mens de to måtene over, alltid implementeres ”heterogent”. I denne oppgaven vil jeg stort sett forholde meg til denne varianten av generiske mekanismer.

Som man kan se ut i fra de tre punktene, varierer det sterkt hvor integrert den generiske mekanismen er i selve programmeringsspråket. Ved hjelp av en tekstlig makrobasert preprosessor, kan man egentlig integrere en enkel generisk mekanisme i alle programmeringsspråk, men samtidig vil det være funksjonalitet man ikke får med.

Ved å integrere mekanismen inn i programmeringsspråket, vil man i tillegg kunne sjekke om pakkene er syntaktisk og semantisk korrekt, og det kan være fordelaktig blant annet når det gjelder prekompilering og generering av maskinkode for bibliotekspakker hvor den konkrete bruken av pakka ikke er kjent.

Når en generisk pakke kompileres til maskinkode, er det forskjellige måter å gjøre dette på. Et skille går blant annet på det som ofte refereres til å kompilere koden på en *heterogen* eller *homogen* måte.

Ved en heterogen implementasjon, som blant annet makrobaserte mekanismer benytter, blir én generisk pakke kompilert ned til forskjellige instanser, én for hver spesifikke bruk, og det blir dermed ikke generert kode for pakka som sådan. Har man en pakke for å sortere tall, blir det generert én kodesekvens når man for eksempel benytter pakka til å sortere heltall og en annen kodesekvens hvis man også benytter pakka til å sortere reelle tall.

En annen måte man kan gjøre det på, men som kun kan benyttes der den generiske mekanismen er en mer integrert del av språket, er at man kompilerer en generisk pakke ned i én felles kode som er fleksibel nok til å kunne brukes for alle parametriseringer. Denne måten refereres gjerne til som en homogen implementasjon.

For at dette skal være mulig, er man nødt til å ha et system som knytter den kompilerte koden opp mot den aktuelle konteksten i ettertid. Den siste tilknytningen kan enten skje f.eks. ved hjelp av en tradisjonell linker, eller i minnet før selve programeksekveringen begynner.

I Java parseres en klassefil av tre uavhengige komponenter i den virtuelle maskinen før bytekoden eksekveres: *Loaderen*, *Verifieren* og *Linkeren*. Se detaljer i [8]. Den siste behandlingen av den generisk kompilerte koden, kan i Java skje i *Loaderen*, som sørger for at bibliotekskласser og andre relaterte klasser blir lest inn i minnet.

2.3 Eksempel på generiske mekanismer

Som jeg var inne på i innledningen, er hovedønsket at det skal være relativt enkelt å skrive programbiter (for eksempel prosedyrer, funksjoner og klasser) som skal kunne brukes i forskjellige kontekster. For å kunne gjøre dette i statisk typede språk, er man nødt til å ha tilgjengelig mekanismer som gjør at man får uttrykt generelle typer som kan brukes i koden.

Jeg var også inne på et eksempel med en sorteringsalgoritme som skulle sortere en liste med heltall, reelle tall og så videre. Under vil jeg presentere hvordan en slik prosedyre kan implementeres ved hjelp av generisk programmering i Ada-95. Etterpå vil jeg også vise at objektorientering, selv uten bruk av generiske mekanismer, også kan løse denne typen problemer et stykke på vei.

2.3.1 Generiske pakker i Ada-95

Ada er et typet språk som ikke er objektorientert. Eller rettere sagt: Ada *var* i utgangspunktet ikke objektorientert, men senere versjoner av språket har innført noen objektorienterte mekanismer. I dette kapittelet vil jeg ikke benytte den objektorienterte delen.

Jeg har valgt å ta med et eksempel skrevet i Ada-95 for å vise hvordan en generisk mekanisme generelt kan fungere. Selv om Ada-95 kanskje er ukjent for mange som har mest er erfaring innen objektorientering, vil jeg ikke gå inn i detaljene i språket. Jeg har heller valgt å vise et eksempel og forklare nærmere hva koden gjør. Utgangspunktet for eksempelet er hentet fra kapittel 5.2 i *"Ada-95 for C and C++ Programmers"* [4].

Siden denne sorteringsalgoritmen skal kunne brukes med forskjellige typer i ulike sammenhenger, er man nødt til å deklarere hvilke variable som er generiske, og som dermed ikke skal bestemmes før i den koden som skal bruke algoritmen.

I Ada blir de generiske typene deklarert før den generiske koden. I denne deklarasjonen setter man opp spesifikasjoner som typene som senere skal kunne benyttes, må oppfylle. Ved hjelp av disse spesifikasjonene kan man statisk sjekke den generiske koden, mens man ved den aktuelle bruken, kun trenger å sjekke om de konkrete variabeltypene som benyttes, oppfyller de spesifikasjonene som er gitt.

I de generiske deklarasjonene kan man henvise til de generiske typene som tidligere er definert, men ikke motsatt. I eksempelkoden som står under, bygger deklarasjonene på hverandre. Som sagt, vil jeg ikke gå inn i detaljene i Ada, men jeg vil nevne et par ting som er relevante i vår sammenheng.

To av de generiske typene er definert med spesifikasjonen `<>`. Dette betyr at typen har et begrenset antall verdier og kan brukes til indeksering av arrayer. I tillegg brukes nøkkelordet `with` som forteller at sorteringsrutinen vil benyt-

te en ekstern relasjonsfunksjon ”større enn”, og man er dermed avhengig av at programmet som skal bruke den generiske koden, må implementere denne.

Etter at man har deklarert de generiske typene, kan man angi selve sorteringsalgoritmen, og man kan da bruke de generiske typene som allerede er deklarert. Koden for selve sorteringen (som kan være egnet for å ligge i et bibliotek), kan være som følger:

```
-- spesifikasjoner for de generiske typene.

generic
    type Index_Type is (<>);
    type Element_Type is private;
    type Element_Array is array (Index_Type range <>)
        of Element_type;
    with function ">" (el1, el2 : Element_Type)
        return Boolean;
procedure Bubble_Sort(The_Array : in out Element_Array);


-- selve sorteringsalgoritmen som kun forholder seg til
-- de allerede definerte generiske typene.

procedure Bubble_Sort(The_Array : in out Element_Array) is
begin
    for Loop_Top in reverse The_Array'Range loop
        for Item in The_Array'First ..
                    Index_Type'Pred(Loop_Top) loop
            if The_Array(Item) >
               The_Array(Index_Type'Succ(Item)) then
                declare
                    Element : Element_Type := The_Array(Item);
                begin
                    The_Array(Item) :=
                        The_Array(Index_Type'Succ(Item));
                    The_Array(Index_Type'Succ(Item)) :=
                        Element;
                end;
            end if;
        end loop;
    end loop;
end Bubble_Sort;
```

Koden for selve sorteringen er altså ikke skrevet slik at den kan benyttes direkte. For å kunne kalle sorteringsalgoritmen, er man først nødt til å ”instansiere” en versjon av den generiske prosedyren, hvor man spesifiserer hvilke typer som skal benyttes for denne spesielle instansen. Under er det vist et eksempel på et program som benytter prosedyren til å sortere en liste med heltall.

```
with Bubble_Sort; -- henter den generiske rutinen inn
                  -- fra biblioteket.
...

subtype Index is Positive range 1 .. 10;
```

```
type An_Array is array (Index range <>) of Integer;

procedure Test_Sort is new Bubble_Sort(Index,
                                       Integer,
                                       An_Array,
                                       ">");
...

Sort_This : An_Array := (5, 6, 7, 1, 3, 2, 4, 10, 9, 8);
Test_Sort(Sort_This);
```

Som man kan se i eksempelet over, er den generiske sorteringsalgoritmen implementert helt generell. I det konkrete brukerprogrammet, har man valgt å sortere en liste med heltall, men med en annen instansiering kan man like gjerne sortere en liste med tekster. Faktisk er prosedyren så fleksibel at man også kan sortere en liste med for eksempel datatyper som representerer personer, og dette er mulig fordi koden tillater programmereren å spesifisere hvilken funksjon som skal brukes ved sammenligning av to konkrete elementer i lista.

I Ada sin implementasjon av generiske typer, kan altså alle pakker sjekkes statisk. Ada har også lagt opp til å bruke en heterogen implementasjon, altså slik at hver ny instansiering blir kompilert til hver sin maskinkode, og det vil dermed ikke være mulig å generere maskinkode for den generelle sorteringsrutinen. Denne mekanismen faller inn under kategori tre hva gjelder språklig integrasjon.

2.3.2 Objektorientert tilnærming

Da objektorienteringen kom med Simula i 1967, var noe av hensikten at objektorienteringen i seg selv ved hjelp av klasser og subklasser, skulle tilby en fleksibilitet i typesystemet som gjorde det mulig å skrive generelle pakker som kunne hentes inn i ulike sammenhenger. Dette tilbyr på mange måter objektorienteringen også, men mekanismen er imidlertid ikke like kraftig på som for eksempel *generics* i Ada.

Under har jeg implementert nesten den samme metoden `bubbleSort` som jeg over viste hvordan man kunne implementere i Ada-95, men denne gangen har jeg benyttet objektorienterte konstruksjoner som er tilgjengelig i Java.

I denne implementasjonen har jeg typet variablene med et interface `Comparable`. Dette er et interface som sørger for at objekter som skal sorteres, inneholder en metode som gjør det mulig å sammenligne to objekter med relasjonen ”større enn”. Det er ikke alle objektorienterte språk som støtter interfacer, men et interface er på en måte det samme som en abstrakt klasse hvor alle metoder er abstrakte. Med andre ord kan man få til nesten det samme ved hjelp av vanlige klasser.

```
interface Comparable {
    boolean greaterThan(Comparable c);
}
```

```
...

void bubbleSort(Comparable[] array) {
    Comparable temp;

    for (int i = 1; i < array.length; i++)
        for (int j = 0; j < array.length - 1; j++)
            if (array[j].greaterThan(array[j + 1]) {
                temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
            }
}
```

Som man kan se av eksempelet, kan man på denne måten få skrevet ganske generelle metoder, men man har likevel ikke like store friheter som man for eksempel har i Ada-95. Sorteringsrutinen i Java har blant annet følgende svakheter:

- Metoden tillater ikke sortering av en tabell bestående av primitiver som `int`, `real` osv.
- Tabeller som skal sorteres, kan kun inneholde objekter av klasser som implementerer interfacet `Comparable`, og det er det mange klasser som ikke gjør; blant annet de standard klassene som leveres med Java.
- Metoden tillater sortering av objekter som kanskje ikke skal kunne sammenlignes. For eksempel vil det fint være mulig å sortere en liste bestående av biler og personer, så lenge begge klassene har implementert det samme interfacet. Med andre ord vil ikke dette bli oppdaget av kompilatoren hvis man gjør en slik feil når man programmerer.

Jeg viste nettopp, i metoden `bubbleSort`, hvordan man kan bruke klasser og subklasser til å representere generelle typer. Denne teknikken kan man benytte videre ved å type alle variable med `Object`. På denne måten kan man sende med objekter av alle typer.

2.4 Generelle typer gjennom det generiske idiom

Java har et enormt klassebibliotek hvor alle klassene henger sammen i en stor struktur. En sentral ting for programmeringen av disse, er at alle klasser i Java, uansett om man spesifiserer det eller ikke, er subklasser av en spesiell klasse `Object`.

Dette kan man utnytte på tilsvarende måte som jeg gjorde i sorteringsalgoritmen. Alle steder hvor man ikke vet typen eller hvor man ikke setter krav til objektene som er parametere, typesetter man med den spesielle klassen `Object`. Her vil det da være mulig å sende med objekter uavhengig av klassen de er generert ut i fra.

Når man siden skal bruke objektet igjen og se på dets innhold, må man spesifisere at objektet faktisk *ikke* er av typen `Object`, men av en eller annen subtype som gjør at man kan få gjort de nødvendige operasjoner. Dette gjør man ved å sette inn en casting til den aktuelle typen/klassen. Denne formen for programmering refereres ofte til som å programmere ved hjelp av *det generiske idiom*.

Problemet med en slik programmering, er at det vil være mange steder i programmet som ikke kan sjekkes statisk. Kompilatoren klarer ikke å sjekke ferdig ”insisterende” kode som castinger. Disse må utføres når programmet eksekverer, og resultatet av dette er at programmering ved hjelp av det generiske idiom, går ut over eksekveringshastigheten. Alle steder hvor man setter inn eksplisitte castinger, blir det utført en typesjekk ved programkjøring.

I objektorienterte språk *må* man altså ikke i like stor grad ha generiske mekanismer for å kunne skrive generelle pakker. Det er snarere for å slippe de stadige castingene og for å få pakkene typet som om de var spesialskrevet for det stedet de skal brukes, at man ønsker seg generiske mekanismer.

Blant annet grunnet de overnevnte problematikkene, tilbyr flere objektorienterte språk generiske mekanismer av ulike typer, og det synes nå å være generell enighet om at Java ville vinne mye på å få en slik mekanisme. Et viktig poeng her er imidlertid at slike mekanismer ikke skal gå ut over hastigheten til programmer som ikke benytter dem. Akkurat dette var også et av kravene Stroustrup stilte når han skulle legge inn støtte for parametriserte klasser og funksjoner i C++ [7].

2.5 Generiske problemområder

Som beskrevet over, blir en del av problemene man vil løse ved generisk programmering, løst ved innføring av objektorienterte begreper. Dog er altså ikke alt løst, og jeg skal i resten av dette kapittelet forsøke å få fram ting som ikke er løst, og antyde hvordan generiske mekanismer kan løse dem. Som en del av dette, vil jeg presentere forskjellige problemstillinger som flere steder blir brukt til å vurdere ”styrken” til forskjellige generiske mekanismer. Flere av dem er hentet fra [8], men noen er også hentet fra annen litteratur.

En del av eksemplene som jeg har skissert, er skrevet i et Java-aktig pseudospråk med en løselig definert generisk mekanisme som forklares etter hvert. Programeksemplene kan ikke kompileres, men er satt inn for å forklare hva jeg ellers beskriver i teksten.

2.5.1 Generell type, List

Ved hjelp av det generiske idiom, er det relativt enkelt å skrive klasser som representerer lister, mengder og stakker. Objektorientert fleksibilitet er nok til å kunne lage en generell liste. Eksempelvis kan man skrive en generell liste i Java med følgende kode:

```
class List {
    void put(Object elem) {...}
    Object get(int pos) {...}
}

...

List l = new List();
l.put(new Person("Ola Knutsen"));
Person p = (Person)l.get(0);
String name = ((Person)l.get(0)).name();
```

Selv om man altså ganske greit kan implementere en generell klasse `List` med bruk av objektorientering, er det likevel ønskelig med en generisk mekanisme. Jeg har allerede vært inne på flere av disse grunnene, og her er en oppsummering av de viktigste:

- Koden blir unødvendig tung å skrive og lese.
- Kompilatoren kan ikke vite at man ønsker å lagre kun `Person`-objekter, og den kan dermed ikke statisk sjekke dette. Hvis man for eksempel setter inn et objekt av en annen klasse, vil ikke dette bli oppdaget før under utførelsen.
- Når man henter ut elementer fra lista, utføres det en "unødvendig" typesjekk ved run-time. Sjekken er unødvendig i den forstand at programmereren trolig vet at det blir hentet ut et `Person`-objekt, og at kompilatoren kunne ha testet det om man bare kunne få "sagt fra" til den.

I eksempelet over er det viktig å skille mellom to typer lister. Hvis man ønsker en liste som skal kunne inneholde forskjellig typer objekter, har man liten nytte av en generisk mekanisme. Man er nødt til å programmere en lignende klasse som det jeg presenterte over. Det er først når man, som antatt over, ønsker en liste som bare skal inneholde én type objekter og kanskje subklasser av denne, at problemet er relevant.

I resten av dette kapittelet vil jeg presentere konkrete eksempler på generiske løsninger, og jeg har valgt å benytte parametriserte klasser der typeparameteren angis i spissparenteser. En lignende syntaks er brukt i C++ og flere andre språk. Igjen gjør jeg oppmerksom på at dette bare er en pseudokode for å klargjøre eksemplene. I kapittel 3 vil jeg presentere flere andre konkrete forslag til generiske syntakser.

Når man implementerer liste-eksempelet ved hjelp av en parametrisert klasse, kan man lett få fortalt kompilatoren at lista kun skal inneholde en bestemt type objekter (og eventuelt objekter av en subtype). Koden vil da kunne være:

```
class List<A> {
    void put(A elem) {...}
    A get(int pos) {...}
}
```

```
...

List<Person> l = new List<Person>();
l.put(new Person("Ola Knutsen"));
Person p = l.get(0);
String name = l.get(0).name();
```

I koden over ser man at man slipper alt som gjelder casting av variable. Siden man forteller kompilatoren at lista kun skal inneholde `Person`-objekter eller objekter av subklasser av `Person`, vil den ikke tillate å sette inn objekter av en annen klasse. Resultatet er at koden kan sjekkes statisk og at man dermed slipper tester under run-time som reduserer eksekveringshastigheten.

2.5.2 Forutsatt metode, HashTable

I eksempelet lenger opp så vi at det var enkelt nok å programmere klasser som implementerer for eksempel lister, selv uten en generisk mekanisme. Dette går relativt enkelt fordi man bare håndterer objektene som blir sendt inn, uten å bry oss om innholdet i hvert objekt.

I det neste eksempelet jeg skal se på, ønsker jeg å kunne uttrykke at alle typer/klasser som skal kunne bli sendt inn som aktuelle parametere, må ha implementert en spesifikk metode. Dette er en forutsetning fordi metoden blir kalt inne i den parametriserte klassen. Eksempelvis vil man ved implementasjon av en `HashTable`, forutsette at objekter som skal bli satt inn, må inneholde en metode `hashCode`.

I utgangspunktet er ikke dette noe stort problem i Java fordi språket støtter bruk av *interfacer*. Ved å type variable med interfacer, vil man kun få lov til å benytte objekter som implementerer det gitte interfacet.

I eksempelet under er det kode for en enkel klasse `HashTable`, som har to generiske parametere: `Key` og `Value`. Den siste kan være en vilkårlig klasse, men for den første er det angitt `"implements Hashable"` som en skranke (spesifikasjon), og det betyr altså at den aktuelle klassen må implementere `Hashable`, og dermed ha metoden `hashCode`. I motsetning til i Ada hvor man i en separat blokk med kode, spesifiserer de generiske typene, har jeg her valgt å sette inn spesifikasjonene ved bruksstedet.

Klasser og interfacer legges i egne filer i Java. Dette er det ikke så enkelt å gjengi i skriftlige dokumenter. Derfor antar jeg i eksemplene framover at det finnes et enkelt `package`-begrep i Java der alle klasser som hører sammen, angis tekstlig under hverandre. Videre bruker jeg `import`-setningen når jeg henter inn de deklarerte klassene.

Bibliotekklassen kan dermed se slik ut:

```
package hashing;

interface Hashable {
    int hashCode();
```

```
}

class HashTable<Key implements Hashable, Value> {
    Value values[];

    void put(Key key, Value value) {
        array[key.hashCode()] = value;
    }

    Value get(Hashable key) {...}
}
```

Brukerapplikasjonen kan være implementert som følger.

```
import hashing;

class Person implements Hashable {
    int hashCode() {...}
}

...

HashTable<Person, int> h = new HashTable<Person, int>();
Person p = new Person("Ola Knutsen");
h.put(p, 123);
int i = h.get(p);
```

Bruk av interfacer som skranker til generiske parametere, er en måte man kan løse problemet på. En annen mulighet er at man kan sette skranker ved hjelp av `"extends <klasse>"` som vil bety av typen som sendes inn, må være en subklasse av den angitte klassen. Eller rettere sagt; når man benytter *extends* i forbindelse med å sette skranker på parametere, menes det at den aktuelle parameteren kan være skrankeklassen selv eller en subklasse i ett eller flere ledd av denne. Den aktuelle parameteren trenger med andre ord ikke å være en direkte subklasse.

Et av problemene som er bundet sammen med bruk av interfacer, er at dette spesifikke interfacet antakeligvis vil være definert i forbindelse med bibliotekspakka for `hashing`. Dermed vil klassen `Person` neppe ha implementert `Hashable`.

Hvis man har klasser som ikke implementerer de påkrevde interfacene, er det mulig å skrive en subklasse som viderefører alle metodekall og som samtidig implementerer interfacene man ønsker. Slike ”wrapper”-klasser kan føre til unødvendig rotete og uryddig kode, og man bør man prøve å holde seg unna dem.

Hvis man skal opprette en ”wrapper”-klasse til `Person`, kan man gjøre det ved å opprette følgende klasse.

```
class PersonWrapper extends Person implements Hashable {
}
```

Siden klassen skal ha lik funksjonalitet som `Person`, trenger ikke den nye klassen å inneholde ny kode, bortsett fra at man sannsynligvis må tilpasse konstruktørmetodene noe. Denne nye klassen kan dermed brukes alle steder hvor man ellers ville brukt `Person`.

Problemet er at man ikke alltid har kontroll på de steder for `Person`-objekter faktisk blir opprettet. Opprettelsen kan blant annet skje i et annet bibliotek. I så tilfelle er man nødt til å lage metoder som omdanner `Person`-objekter til `PersonWrapper`-objekter, noe som igjen gjør koden mer uryddig. Jeg kommer tilbake til disse problemstillingene i kapittel 3.

2.5.3 Forutsatt metode med gitt typeparameter, Prioritetskø

I eksempelet med `HashTable` tok jeg for meg problemet som oppstår når et objekt som blir sendt med som parameter, må ha implementert en bestemt metode. Denne problematikken kan man ta et hakk videre. I tillegg til at et objekt må ha implementert en bestemt metode, kan man tenke seg at denne metoden igjen må ha en parameter av en generisk type.

Man må altså kunne spesifisere en generisk parameter slik at den aktuelle klassen må inneholde en metode som tar en parameter av en bestemt generisk type, ofte samme type som klassen selv. Dette kan man få til ved for eksempel å ha mulighet til å opprette generiske interfacer på samme måte som med klasser.

Agesen bruker et eksempel med en prioritetskø i sin artikkel [8]. En prioritetskø er en liste hvor rekkefølgen av elementene blir ordnet ut i fra en lineær relasjon, for eksempel ”mindre enn”.

Ved implementasjon av en prioritetskø, er det en forutsetning at alle objekter som skal settes inn i køen, har implementert en binær relasjonsmetode. Et objekt må kunne sammenlignes med et objekt av samme type, så metoden må derfor ha en parameter av samme type som seg selv. En prioritetskø kan for eksempel skrives med følgende kode hvor jeg har benyttet et parametrisert interface. I biblioteket vil følgende være tilgjengelig:

```
package priorityQueue;

interface Comparable<I> {
    boolean lessThan(I other);
}

 class PriorityQueue<T implements Comparable<T>> {
    T queue[];

    void insert(T elem) {
        if (elem.lessThan(queue[i]))...
        ...
    }
}
```

Biblioteket kan benyttes i en applikasjon som følger:

```
import priorityQueue;

class Prime implements Comparable<Prime> {
    boolean lessThan(Prime other) {...}
}

PriorityQueue<Prime> pq = new PriorityQueue<Prime>();
pq.insert(new Prime(5));
```

Når jeg i kapittel 3 skal se på forskjellige forslag til generiske mekanismer, vil jeg bruke dette eksempelet til å se om forslagene klarer denne type komplisert spesifikasjon, og om det eventuelt kan gjøres på en grei måte.

2.5.4 Utvidelse av en klasse, Mix-In

Den siste og mest kompliserte problemstillingen jeg skal se på i dette kapittelet, er også hentet fra [8]. Her er ideen at man har forskjellige klasser som alle implementerer visse metoder, for eksempel `lessThan` og `equal`, der den aktuelle metodeparameteren typisk er typet til å angi klassen selv.

Om de angitte metodene i de forskjellige klassene har samme logiske betydning (her størrelsesrangering av objektene, i en betydning som kan variere fra klasse til klasse), kan det være aktuelt å utvide settet av metoder på en "felles logisk måte", med for eksempel `greaterThan` og `notEqual`, hvor disse bygger på metodene som allerede er gitt.

Et eksempel på en slik klasse, kan være en klasse kalt `SimpleRelations`. Denne klassen inneholder de omtalte "grunnmetodene", og en subklasse av denne, her kalt `AdvancedRelations`, kan dermed utvide settet med de nye metodene. Et slikt eksempel kunne ta seg ut slik:

```
class SimpleRelations {
    boolean lessThan(SimpleRelations other) {...}
    boolean equal(SimpleRelations other) {...}
}

class AdvancedRelations extends SimpleRelations {
    boolean greaterThan(SimpleRelations other) {
        return (other.lessThan(this) || notEqual(other));
    }

    boolean notEqual(SimpleRelations other) {
        return !other.equal(this);
    }
}
```

Men denne utgaven av `AdvancedRelations`, virker selvfølgelig bare for akkurat den angitte klassen `SimpleRelations`. Spørsmålet blir da om man kan skrive en generisk utgave av `"AdvancedRelations"` som kan ta den aktuelle utgaven av `"SimpleRelations"` som en generisk parameter, og "produsere" en riktig typet klasse tilsvarende `AdvancedRelations`.

Om man vil gjøre det etter samme mønster som for prioritetskøen, kunne dette løses med følgende generiske interface og klasse, hvor interfacet `LessAndEqual` spiller rollen til `SimpleRelations` og klassen `Relations` rollen til `AdvancedRelations`. Disse elementene kan samles i følgende passelige pakke:

```
package relations;

interface LessAndEqual<I> {
    boolean lessThan(I other);
    boolean equal(I other);
}

class Relations<C implements LessAndEqual<C>> extends C {
    boolean greaterThan(Relations<C> other) {
        return (other.lessThan(this) && notEqual(other));
    }

    boolean notEqual(Relations<C> other) {
        return !other.equal(this);
    }
}
```

Den generiske klassen `Relations` uttrykker følgende: For det første tar den en klasse som parameter, samtidig som koden sier at klassen `Relations` selv er en subklasse av den som sendes inn. I tillegg kreves det at klassen som sendes inn som parameter, har implementert et interface hvor parameterklassen er sendt med som parameter her også. Dette gjøres for å kreve binærmetoder på samme måte som i eksempelet rundt prioritetskøen.

Ved hjelp av den generiske bibliotekskassen skrevet over, kunne man opprettet en klasse lignende `AdvancedRelations` med følgende applikasjonskode:

```
import relations;

class SimpleRelations implements
          LessAndEqual<SimpleRelations> {
    boolean lessThan(SimpleRelations other) {...}
    boolean equal(SimpleRelations other) {...}
}

Relations<SimpleRelations> advancedRelations =
    new Relations<SimpleRelations>;
```

Den nye variabelen kan benyttes på samme måte som om den var typet med `AdvancedRelations`, og man ser herved hvor fleksibel den generiske klassen er.

Jeg skal i neste kapittel se på om de foreslåtte generiske mekanismene, kan uttrykke noe såpass komplisert som klasser lignende `Relations`.

2.6 Andre problemtyper

I tillegg til de litt større problemstillingene som jeg skissert over, vil det være flere momenter jeg kommer til å trekke inn i en helhetsvurdering av hver enkelt generisk mekanisme. For det første er det viktig å se på hva forfatterne av de forskjellige artiklene selv har satt seg selv som mål. Hvor langt mener de at de går, og eventuelt hvilke tidligere arbeider og erfaringer bygger de på? Det vil også være av klar fordel om forslagsstilleren har implementert og testet ut mekanismen i praksis, i stedet for bare å vite om hvordan metoden fungerer i teorien.

Noen andre elementer man bør ta med i vurderingen er følgende:

Gjentakelse av typeparameter: Hvordan gis spesifikasjonen for typeparameterne? De forskjellige mekanismene har ganske forskjellige filosofier på hvordan dette bør gjøres. Enkelte krever at man hele tiden må repetere de generiske typene nedover i programmet, mens blant annet GJ som presenteres i [6] prøver å være litt mer ”intelligent”. Her deklareres typene i starten, og ellers prøver kompilatoren for generiske metoder (ikke for generiske klasser) å skjønne hva programmereren har ment ut fra bruken. Dette fører til visse begrensninger, men på den annen side er det mer behagelig fordi man slipper å gjenta seg selv hele tiden.

Primitive typer: Java, som de fleste objektorienterte språk, støtter separatkompilering av klasser. Dette vil si at man kompilerer ned ferdig skrevne klasser uten å ha plassert dem inn i en kontekst. Når man gjør dette, tilordner man variabler og funksjoner relative minneadresser, og disse blir endret når man linker sammen de prekompilerte klassene. Når man kompilerer generiske klasser, blir dette en del vanskeligere. Kompilatoren vet ikke lenger vet hvor stor plass hver variabel okkuperer av minnet siden for eksempel typen `double` okkuperer større plass av minnet enn typen `char`. Hvis man forutsetter at alle generiske typer kun kan være pekere, er ikke problemet like stort fordi pekere har en bestemt størrelse. Jeg vil derfor også vurdere hvor godt mekanismen i hver enkelt artikkel takler primitiver som `int`, `real`, `char` osv.

Ekte typer: Et annet viktig moment er om de generiske typeparametrene kan brukes akkurat som vanlige klasser. Det vil si at man for eksempel skal få lov til å subklasse de generiske typene. I C++ bruker man *templates* for å skrive generisk kode. Disse er mer statiske og kan ikke benyttes på samme måte som for eksempel typene som brukes i Beta, *virtuelle patterns*. I Beta kan man blant annet subklasse en generisk type som blir sendt inn i en ytre klasse.

Integrering: Innledningsvis var jeg inne på viktigheten av biblioteksbruk. En spesiell utfordring vil da være å integrere enkelt og fleksibelt gamle klasser som kanskje har løst generisk typing gjennom *det generiske idiom*, sammen med nye klasser som kan ha benyttet for eksempel parametriserte klasser. Det mest ideelle er å ha en måte som tillater bruk av

både generiske og ikke generiske biblioteker samtidig, noe som gjør det greit å få kompilert og kjørt programmer under en omleggingsfase, der man én for én legger om bibliotekskklassene fra det generiske idiom til for eksempel generiske parametere.

3 Forslag til generiske typer i Java

I forrige kapittel presenterte jeg en del ønsker og problemtyper som man kan se for seg løst i programmeringsspråk ved hjelp av generiske mekanismer. Nå skal jeg gå nærmere inn på de generiske mekanismene som er foreslått i de artiklene jeg har lest. Under denne gjennomgangen vil jeg samtidig se på i hvilken grad de kan løse eksemplene som jeg skisserte i forrige kapittel.

Som jeg nevnte tidligere, har det vært viktig for forfatterne av artiklene å holde seg innenfor rammene av den nåværende standard for JVM-er. Dette har flere steder lagt begrensninger på de foreslåtte generiske mekanismene. I mine vurderinger og forslag skal jeg imidlertid ikke begrense meg på samme måte. Ved gjennomgangen av artiklene, har jeg derfor ikke fokusert spesielt på integreringen med nåværende Java.

Flere av artiklene går også ganske detaljert inn på hvordan deres foreslåtte mekanisme kan implementeres i en Java-kompilator. Dette ligger noe utenfor fokus for denne oppgaven, og jeg har derfor ikke brukt mye plass på dette.

De artiklene jeg skal se på er:

- *"Genericity in Java with Virtual Types"* [10], skrevet av Kresten Krab Thorup.
- *"Adding Type Parameterization to the Java Language"* [8], skrevet av Ole Agesen, Stephen N. Freund og John C. Mitchell.
- *"Parameterized Types for Java"* [15], skrevet av Andrew C. Myers, Joseph A. Bank og Barbara Liskov.
- *"Making the future safe for the past: Adding Genericity to the Java Programming Language"* [6], skrevet av Gilad Bracha, Martin Odersky, David Stoutamire og Philip Wadler.

I slutten av kapittelet vil jeg oppsummere og komme med noen tanker om hvordan forslagene eventuelt kunne kombineres og videreføres.

3.1 Virtuelle typer, Thorup

Kresten Krab Thorup beskriver i sin artikkel *"Genericity in Java with Virtual Types"* [10] en generisk mekanisme for Java som i utgangspunktet er ganske lik Beta sin implementasjon av virtuelle patterns. Forfatteren presenterer en fullverdig generisk mekanisme som kanskje ikke er like kraftig som enkelte av de andre mekanismene, men som på den annen side ikke har behov for å

definere et eget begrep ”generisk klasse” (som må ”instansieres” før det blir en ”vanlig” klasse) slik de fleste andre forslag må.

3.1.1 Generell type, List

Grunntanken bak Thorup sine virtuelle typer, er at man i en klasse navngir en type (som må være en klasse) ved hjelp av nøkkelordet `typedef`. I subklasser til den klassen hvor denne definisjonen er gjort, kan man så komme med en ny slik definisjon for det samme navnet, men denne gangen med en annen type, som må være subklasse av den tidligere. Vitsen er da at man i objekter av superklassen skal bruke den første definisjonen, mens man i hele objektet av subklassen skal bruke den nye. Den ”dypeste” definisjonen skal alltid gjelde for *hele* objektet.

For å illustrere mekanismen, presenterer jeg et forslag til hvordan klassen `List` fra kapittel 2 kan skrives. Først skriver man en generell klasse `List` hvor pekere til listeelementene er typet med klassen `Object`, men hvor man hele tiden henviser til det definerte typeordet `ElemType`.

```
class List {
    typedef ElemType as Object;

    void put(ElemType elem) {...}
    ElemType get(int pos) {...}
}
```

Denne klassen kan nå benyttes slik den er om man ønsker en liste som kan inneholde alle slags objekter. Men om objektene som skal inn i lista er av én bestemt og kjent type, har man fortsatt de samme problemene som ved det generiske idiom, men disse kan løses ved å lage en subklasse av `List` med en ny definisjon av `ElemType`. Hvis man for eksempel ønsker seg en liste som kun kan inneholde objekter av en gitt klasse `Person`, oppretter man en subklasse av `List` hvor `ElemType` nå redefineres til å angi klassen `Person`. Resten av klassen arves som vanlig.

```
class PersonList extends List {
    typedef ElemType as Person;
}
```

Denne klassen kan nå brukes slik:

```
PersonList l = new PersonList();
l.put(new Person(“Ola Knutsen”));
// Trenger nå ikke noe casting i det følgende:
Person p = l.get(0);
String name = l.get(0).name();
```

Siden man i eksempelet over, vet at `l` (minst) er et `PersonListe`-objekt, slik at `ElemType` (minst) er av typen `Person`, trenger man ikke lenger å benytte casting når man henter ut objekter eller når man prøver å aksessere attributter

direkte i et objekt. Det vil med andre ord i større grad være mulig å statisk sjekke et slikt program.

Det vil selvfølgelig også være mulig å lage en ytterligere subklasse av `PersonList` (og dermed `List`), hvor `typedef` for eksempel settes til å angi en klasse `Student`, hvor `Student` da må være en subklasse av `Person`.

Den generiske mekanismen til Thorup benytter en kovariant tankegang, noe som innebærer at når man lager en subklasse og samtidig ønsker å endre `typedef`, er man nødt til å sette `typedef` til å angi en subtype av den allerede angitte typen. I eksempelet over er det altså i `PersonList` lov å sette `ElemType` til å henvise til klassen `Person` fordi `Person` er en subklasse av `Object`. Denne regelen er nødvendig for at det som er skrevet i superklassen fremdeles skal fungere, og det vil i de fleste tilfeller også falle naturlig og ikke virke hemmende på programmeringen. Denne kovariante tankegangen understreker Thorup som en viktig egenskap ved hans mekanisme.

3.1.2 Forutsatt metode, HashTable

I de neste eksemplene vil det melde seg et behov for å kunne sette skranker på en typedefinisjon, og dette støtter Thorup sin mekanisme på en grei måte. Generelt kan en `typedef`-setning være på formen

```
typedef A as B, C, D = E;
```

hvor setningen enten kan være definert som `final`, `abstract`, `public`, `protected` eller `private` (sammensatt etter vanlig Java-konvensjon). `A` er her navnet på bindingen, mens `B`, `C` og `D` er skranker som `A` må oppfylle, altså være en subklasse av eller implementere. Skrankene kan være definert ved én klasse og/eller én eller flere interfacer. `E` er klassen som brukes ved instansiering av den virtuelle typen, og `E` må også oppfylle skrankene som er gitt gjennom `B`, `C` og `D`.

I mange tilfeller vil ikke `typedef`-setningen være spesifisert med andre skranker enn klassen `E` selv, og Thorup har derfor innført en kortvariant for dette, der man i stedet for å skrive `E = E`, kun angir skranken. En typisk definisjonen kan dermed være på den formen vi har sett over, for eksempel:

```
typedef Elem as Hashable;
```

Ved hjelp av disse mulighetene for å spesifisere skranker, vil det ikke være noe problem å opprette en generell klasse `HashTable` med funksjonalitet som i kapittel 2.5.2. Dette kan gjøres ved å spesifisere `Key` med interfacet `Hashable`, mens `Value` bare spesifiseres til `Object`. Altså slik:

```
class HashTable {
    typedef Key as Hashable;
    typedef Value as Object;
    ...
}
```

Videre kan man lage en mer spesifikk klasse ved å subklasse `HashTable` og samtidig endre `typedef`-henvisningene. I eksempelet under går jeg ut i fra at man har klassen `Person` (fra kapittel 2) som implementerer interfacet `Hashable`.

I eksempelet i kapittel 2 er verdien som sendes inn av typen `int`, men siden Thorup kun jobber med klasser, er det ikke lov å la `typedef` henvise til en primitiv type. Derfor er man nødt til å benytte "wrapper"-klassen `Integer` i stedet. Klassen kan da representeres med følgende kode:

```
class PersonIntegerHashTable extends HashTable {
    typedef Key as Person;
    typedef Value as Integer;
}
```

Denne klassen kan benyttes på tilsvarende måte som jeg har vist for `List`, og klassen `HashTable` kan ved hjelp av skrankene sjekkes statisk uten å vite noe om senere bruk.

3.1.3 Forutsatt metode med gitt typeparameter, Prioritetskø

I sin generiske mekanisme, innfører Thorup en ny "dynamisk" type kalt `This` (ikke det samme som `this`) som automatisk vil være tilgjenglig i alle klasser. Dette er en type som hele tiden vil henvise til klassen av det objektet man befinner seg inni. Kompilatoren må imidlertid være "konservativ", og de fleste steder vil den anta at `This` angir den klassen den er skrevet i.

Hvis man for eksempel i en klasse `Link` typer en variabel med `This`, vil denne endres i et objekt av en subklasse av `Link`. Et enkelt listesystem illustrerer bruken av `This` ganske godt.

```
class Link {
    This next;
}

class Person extends Link {
    String name;
}
```

Klassen `Person` kan nå benyttes til å opprette en liste med personer:

```
Person p = new Person("Ola Hansen");
...
// Følgende vil nå fungere greit uten casting fordi "next" i
// et Person-objekt har typen "Person".
String name = p.next.name;
```

Ved hjelp av denne `This`-mekanismen, kan man også lage subklasser av `Person`, og i slike objekter vil da `This` referere til den nye klassen. Likevel er det et lite problem. Det er vanskelig å implementere `This` på en slik måte

at koden alltid kan sjekkes statisk. `This` er i utgangspunktet en delvis dynamisk type, og følgende eksempel illustrerer et av problemene.

```
class Student extends Person {
    int courseNr;
}
...
Person p = new Student("Ola Hansen");
...
// Kompilatoren vil ikke kunne sjekke følgende tilordning:
p.next.coursnr = 5;
```

I dette eksempelet vil en kompilator neppe godkjenne programmet, mens den i andre tilfeller vil legge inn tester under utførelsen, for eksempel ved:

```
p.next = new Person();
```

Problemet i dette eksempelet er at `p` enten kan peke til et `Person`-objekt eller et `Student`-objekt. I det første tilfellet er operasjonen ok, mens i det andre er den ulovlig. Kompilatoren må derfor legge inn en test som gjøres under utførelsen, og disse testene er en klar ulempe ved Thorups forslag.

Det skal legges til at tilsvarende konstruksjoner lett kan oppstå med virtuelle klasser generelt, selv uten bruk av `This`. Men det skal også nevnes at denne typen problemer kan til en viss grad unngås, om språket gir mulighet til å definere såkalte *endelige* bindinger.

Likevel kan `This` være nyttig i mange tilfeller, og i det tredje eksempelet som jeg bruker for å vurdere styrken til de generiske mekanismene, kan bruk av `This` forenkle koden rundt implementasjon av prioritetskøen fra kapittel 2. Her er det nødvendig at man klarer å uttrykke binærmetoder som senere kan redefineres med andre typeparametere. Ved å type parameteren til binærmetoden med `This`, oppnår man greit at typen hele tiden vil være den samme som den omkringliggende klassen.

På biblioteket kan man legge inn følgende kode for prioritetskøen:

```
package priorityQueue;

interface Comparable {
    boolean lessThan(This other);
}

class PriorityQueue {
    typedef T as Comparable;

    T queue[];

    void insert(T elem) {
        if (elem.lessThan(queue[i]))...
        ...
    }
}
```

I brukerprogrammet kan man nå opprette en klasse som implementerer `Comparable`, samtidig som man subklasser `PriorityQueue` og endrer `typedef` til å representere den nye typen. Brukerprogrammet vil da kunne være:

```
import priorityQueue;

class Prime implements Comparable {
    boolean lessThan(This other) {...}
}

class PrimePriorityQueue {
    typedef T as Prime;
}
```

Det er her viktig at kompilatoren per definisjon tolker det slik at `lessThan` i `Prime` er en redefinisjon av `lessThan` i interfacet `Comparable`, altså at `This` i denne sammenheng regnes som samme type i sub- og superklassen.

3.1.4 Utvidelse av en klasse, Mix-In

Forfatterne av [8] påstår i sin artikkel at det er flere foreslåtte generiske mekanismer som ikke klarer å løse eksempelet med Mix-in. Thorup sitt forslag til virtuelle typer er en av mekanismene som i utgangspunktet ikke er kraftig nok til enkelt å kunne uttrykke det. Under vil jeg vise hvor mekanismen, i hvert fall i første omgang, kommer til kort.

For å få til det samme som i 2.5.4 kan man forsøke å ta utgangspunkt i de forrige eksemplene, bortsett fra at `LessAndEqual` her må være en klasse siden `Relations` skal være en utvidelse av denne. Man kan så forsøke å implementere bibliotekskassene som følger:

```
abstract class LessAndEqual {
    abstract boolean lessThan(This other);
    abstract boolean equal(This other);
}

class Relations extends LessAndEqual {
    typedef C as LessAndEqual;

    // inneholder de nye metodene basert på de eksisterende
    ...
}
```

Problemet oppstår når man i et brukerprogram ønsker å opprette en konkret klasse, som skal være en subklasse til `LessAndEqual` samtidig som den skal være en superklasse til `Relations`. Det er her umulig å definere klassen `Relations` slik at den blir en subklasse til en konkret brukerklasse som ikke vil være kjent med kompilering av biblioteket. Dermed virker ikke den innfallsvinkelen.

Likevel finnes det en mulighet som gjør at man kanskje kommer et skritt nærmere en løsning. Java støtter bruk av indre klasser (klasser som innehol-

der andre klasser). Ved å plassere klassene `LessAndEqual` og `Relations` på biblioteket inne i en omkringliggende klasse `RelationsContainer`, åpner det seg en mulighet som avhenger av om man i Thorups forslag tillater såkalte ”virtuelle superklasser” (må ikke blandes sammen med en lignende betegnelse i C++). Vi kunne da legge følgende på biblioteket:

```
class RelationsContainer {
    typedef C as LessAndEqual;

    abstract class LessAndEqual {
        // som definert over.
        ...
    }

    // C blir her en virtuell superklasse for "Relations".
    class Relations extends C {
        // som definert over.
        ...
    }
}
```

I brukerprogrammet må man også opprette en klasse som inneholder de andre klassene man behøver. Denne klassen, som her er kalt `AdvancedRelationsContainer`, skal være en subklasse av `RelationsContainer`. Brukerprogrammet vil da være:

```
class AdvancedRelationsContainer extends RelationsContainer {
    typedef C as SimpleRelations;

    class SimpleRelations extends LessAndEqual {
        // inneholder de faktiske definisjonene av
        // "lessThan" og "equal".
        ...
    }

    class AdvancedRelations extends Relations {
    }
}
```

Eksempelet over kan være en tilnærming av en løsning på problemet, men i tillegg til å være noe ”ad hoc”, er jeg heller ikke sikker på om dette er kode som er gyldig etter Java og Thorup sine spesifikasjoner. Blant annet er jeg skeptisk til at klassen `AdvancedRelations` er en subklasse av `Relations` som igjen er en subklasse av `C`. Klassen `C` blir jo først definert i brukerprogrammet. Med andre ord vil jeg si at dette kan være et forslag til en tilnærming, men ikke en direkte løsning på problemet.

Thorup sin artikkel er ikke så spesifikk at den sier om dette er lov eller ikke, men man kan merke seg at det i programmeringsspråket Beta i utgangspunktet er mulig å lage tilsvarende konstruksjoner, men der har de eksplisitt valgt å forby dem da det kan føre til kode som er sen å eksekvere, faktisk selv om mekanismen ikke brukes.

3.1.5 Sammendrag

De virtuelle typene som Thorup presenterer, faller godt inn i Java sin nåværende standard. I utgangspunktet utvider Thorup bare mulighetene innenfor Java sine klasser slik brukerne allerede kjenner dem.

Siden virtuelle typer kun opererer med reelle klasser, gir det nesten ikke mening å snakke om en homogen eller heterogen implementasjon. Implementasjonen blir helt naturlig homogen med de fordeler og ulemper det medfører.

Virtuelle typer tar som sagt kun utgangspunkt i klassebegrepet, og dette utelukker dermed støtte for primitive typer. Dette er ikke kritisk siden Java i tillegg til forskjellige primitive typer, også har standard klasser som representerer hver av dem, men det er likevel en ulempe siden man av og til kan ha behov for å jobbe med de primitive typene direkte.

Den største ulempen med virtuelle typer, er antakeligvis at mekanismen ikke har mulighet til å bli like kraftig som for eksempel parametriserte typer. Modellen bak mekanismen er kanskje teoretisk bedre fundert, men samtidig medfører dette at man blant annet ikke klarer å uttrykke eksempler som Mix-in.

3.2 Parametriserte typer, Agesen mfl.

Ole Agesen, Stephen N. Freund og John C. Mitchell har skrevet artikkelen *"Adding Type Parameterization to the Java Language"* [8]. I denne artikkelen diskuterer forfatterne deres inntrykk og syn på mangelen av generiske typer i Java, samtidig som de presenterer et komplett forslag til parametriserte typer.

Forslaget til Agesen har mange fellestrekk med lignende forslag som har vært presentert før, særlig innenfor andre språk. Blant annet bruker Eiffel [21] parametriserte typer som har mange fellestrekk med Agesens, og her kan man blant annet også spesifisere generiske parametere med krav om at aktuellparametere er en subklasse av en spesifikk klasse.

I Eiffel kan man for eksempel opprette en generisk klasse `HashTable` med krav om at en aktuellparameter skal være en subklasse av `Hashable`, med følgende kode:

```
class HashTable[Element -> Hashable]
    ...
end
```

For de fleste problemtypene jeg presenterte i kapittel 2, illustrerte jeg der også en løsning ved hjelp av en mekanisme som tar utgangspunkt i nettopp de parametriserte typene fra [8]. Derfor gjentar jeg ikke løsningene i dette kapittelet, men diskuterer litt mer generelt om den tekniske løsningen.

3.2.1 Generelt om den generiske mekanismen

Artikkelen [8] sitt forslag til parametriserte klasser, ligner syntaktisk en del på *templates* fra C++. Det hele bygger på at man skriver en ”halvferdig” klasse (kan kalles en generisk klasse) som tar en eller flere formelle klasseparametere, og klassen blir først fullstendig når den instansieres og man samtidig sender med spesifikke klasser som aktuelle parametere. Det er først da klassen blir ”fullverdig” og kan benyttes på lik linje som andre uparametriserte klasser.

I tillegg til å legge inn støtte for parametriserte klasser, har de også innført parametrisering på interfacer. Parametriserte interfacer, kan blant annet gjøre det enklere å sette krav til binærmetoder, hvor typen vil variere fra bruk til bruk.

Da forfatterne designet sitt forslag, tok de utgangspunkt i at følgende krav skulle oppfylles:

- De parametriserte typene skulle være relativt enkle å forstå, slik at programmerere kunne ta dem i bruk uten dyptgående kunnskap om hvordan de fungerer.
- De ønsket å forhindre innføring av nye begreper i Java, så fremt det var mulig.
- De parametriserte typene skulle øke muligheten til statisk sjekking og til å redusere bruk av castinger (så lenge det var mulig), for på denne måten å kunne oppdage programmeringsfeil så tidlig som mulig og dermed også redusere kostnadene ved programeksekvering.
- Mekanismen skulle støtte separatkompilering slik at man, i tråd med Java sin idé, statisk skulle kunne sjekke separate enheter for feil selv uten å kjenne de aktuelle parameterne.

For å gjøre det mulig å statisk sjekke den generiske koden, har forfatterne lagt inn støtte for å kunne sette skranker på (eller ”spesifisere”) de generiske parameterne. Dette gjør de i utgangspunktet ved å la brukeren benytte de allerede kjente begrepene `implements` og `extends`, hvor de i denne sammenheng skal bety:

- `A implements I`: Parameteren `A` er enten en klasse som implementerer `I`, en subklasse i en eller flere steg av en slik eller et interface som i null eller flere steg har `I` som et superinterface.
- `A extends B`: Parameteren `A` er en subklasse av `B` i null eller flere.

Disse to skrankene kan enten brukes hver for seg eller kombineres hvis det er ønskelig. Det vil derfor være fullt lovlig for eksempel å definere følgende klasse:

```
class C<A implements I, J extends B> {...}
```

I utgangspunktet gjør disse skrankene, på samme måte som for Ada, det mulig å statisk sjekke den generiske klassen separat, og generelt er det ikke så mye mer man trenger å tenke på når man programmerer ved hjelp av parametriserte typer. Likevel er det at par konstruksjoner man må være oppmerksom på, og som gjør at parametriserte klasser ikke er helt trivielle. Ta for eksempel følgende klasse:

```
class C<A> {
    void m(A s) {...}
    void m(String s) {...}
}
```

Dette vil være en gyldig klasse for alle parametere bortsett fra for klassen `String`. Hvis man sender med en `String`-parameter, vil klassen inneholde to metoder med samme grensesnitt, noe som vil være flertydig og som derfor er ulovlig i følge standarden til Java. Agesens parametriserte typer tillater derfor ikke at klassen `C` instansieres med typen `String`.

Mekanismen tillater at man kan bruke primitive typer som parametere til generiske klasser; men selvfølgelig ikke der det forutsettes at parameteren enten implementerer et interface eller utvider en klasse. Hvis dette skal gjøres, er man nødt til å bruke de tilsvarende klassene for de primitive typene som leveres med Java.

Som jeg viste i kapittel 2, tillater Agesens mekanisme at man henviser til en generisk parameter, som en del av spesifikasjonen av de generiske parameterne. Eksempelvis:

```
class Relations<C implements LessAndEqual<C>> extends C {
    ...
}
```

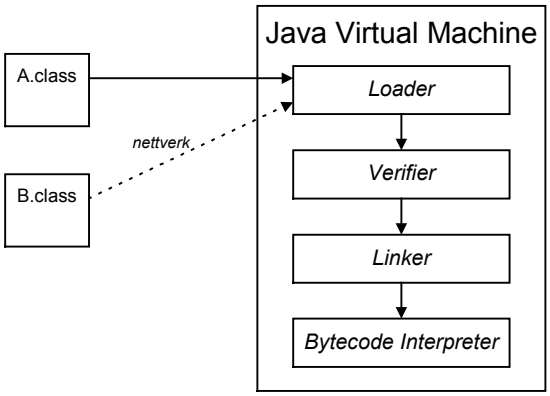
Dette gjør at man får ganske fleksible muligheter ved deklarering av spesifikke parametere. Det teoretiske grunnlaget (kalt *F-Bounded Polymorphism*) for at dette blir konsistent, er blant annet gitt i artikkelen *"F-Bounded Polymorphism for Object-Oriented Programming"* [25].

3.2.2 Integrering av mekanismen i eksisterende JVM

Forfatterne som har foreslått denne mekanismen, har også tatt utgangspunkt i at den skal være bakoverkompatibel med tanke på gamle JVM-standarder. De har imidlertid vært nødt til å gjøre en liten forandring på *loaderen* i den virtuelle maskinen, men denne forandringen får ingen følger for programmer skrevet for tidligere JVM-er.

En JVM er bygget opp av fire forholdsvis atskilte komponenter, og *loaderen* er den første komponenten som en klassefil parseres av. Denne leser inn klassefila og henter eventuelt inn andre klasser som programmet er avhengig av. Se Figur 3.1.

Grunnstrategien til [8] er å kompilere en parametrisert klasse ned til en utvidet form for klassefil. I tillegg til alt som vanligvis lagres i klassefila, inneholder den utvidete versjonen også informasjon om parametere og skranker, og denne informasjonen er lagt inn i et ubrukt helt i klassefilformatet.



Figur 3.1 Oversikt over Java sin virtuelle maskin.

Når den virtuelle maskinen leser inn en instans av en parametrisert klasse eller interface, vil en preprosessor i *loaderen* transformere den til en normal klassefil. Med andre ord blir dette en heterogen mekanisme sett fra JVM-en under utførelsen, hvor `List<Person>` representeres med egen bytekode som er forskjellig fra koden for `List<Car>`. Selve bytekoden for en generisk klasse produseres imidlertid bare én gang, og dette gjøres uten å kjenne de aktuelle generiske parametere.

Agesen og kollegene har gjennomført noen ytelsestester med sin prototyp av den nye kompilatoren og den utvidete JVM-en, og resultatet er dels positivt. På den ene siden vil et program bruke noe lenger tid mens det leses inn i minnet på maskinen, fordi *loaderen* må modifisere de generiske klassene, men etterpå eksekverer programmer som bruker typeparametriserte klasser raskere enn de tilsvarende programmer som bygger på det generiske idiom, blant annet fordi en del typetester er fjernet.

Forfatterne utførte blant annet en test på et program som hadde implementert `HashTable` beskrevet tidligere, og her opplevde de en ytelsesforbedring på 2-3% i forhold til om programmet skulle vært skrevet ved hjelp av det generiske idiom.

3.2.3 Sammendrag

Agesen og kollegene har i sin artikkel presentert en nokså rett fram, men kraftig generisk mekanisme som tar mye utgangspunkt i templates fra C++, men med full typespesifisering med den fleksibilitet som blant annet ”F-Bounded Polymorpism” gir. De kan dermed statisk sjekke og separatkompilere de generiske klassene.

De har valgt å ikke innføre flere nye begreper enn nødvendig, og forholder seg derfor til de kjente nøkkelordene *extends* og *implements* ved skrankesetting, selv om de altså er gitt en noe "utvidet" betydning. Til forskjell fra Thorups forslag om virtuell typer, har de imidlertid (som de fleste andre forslag) måttet innføre begrepet "generiske klasse" i tillegg til de vanlige klassene.

Forfatterne har som sagt tatt mye utgangspunkt i templates fra C++, med de har valgt å benytte en implementasjon som er noe midt mellom heterogen og homogen. De har også valgt at forskjellige instanser av en parametrisert klasse, ikke står i noe super-/subklasseforhold til hverandre. Klassen `List<Person>` en dermed ikke en subtype av `List<Object>` selv om `Person` er en subtype av `Object`. Agesen har med andre ord ikke vektlagt den kovariante tankegangen som Thorup mente var viktig.

3.3 Parametriserte typer, Myers mfl.

Artikkelen *"Parameterized Types for Java"* [15] legger fram et forslag til innføring av "parametrisert polymorfisme" i Java, som er nok en mekanisme for å skrive generiske klasser og interfacer. Forfatterne presenterer også en fullverdig implementasjon, og de har utviklet en prototyp på en utvidet kompilator og bytekode-interpreter som støtter deres nye mekanisme.

I utgangspunktet har mekanismen mange fellestrekk med den som ble presentert i forrige avsnitt [8], men i stedet for å sette skranker på klasseparametere ved hjelp av nøkkelordene *implements* og *extends*, bruker [15] en annen form for skranker kalt *where-clauses*, en mekanisme som har tråder tilbake til språket CLU [26] og [29]. Denne gjør det mulig å spesifisere skranker med krav direkte til metoder i stedet for til hele klasser og/eller interfacer.

Forfatterne har valgt å ta utgangspunkt i en tidligere kjent mekanisme med navn Theta-mekanismen [19] og [20] ved utforming av deres parametriserte typer for Java. Dette har de blant annet valgt å gjøre da denne mekanismen støtter fullstendig separat typesjekking av moduler.

3.3.1 Generelt om den generiske mekanismen

Forfatterne forteller i starten av artikkelen at de hadde som utgangspunkt å være så konservative som mulig, i den forstand at de ønsket å benytte en allerede fungerende mekanisme og kun gjøre så små endringer på denne som mulig. Blant annet så de på templates i C++, men denne mekanismen støttet ikke forfatterne og Java sitt mål om komplett typesjekking uten kjennskap til de aktuelle typeparameterne. I tillegg ville de lage en mekanisme som tillot en homogen implementasjon; noe templates ikke tillater.

Som jeg nevnte innledningsvis bruker denne mekanismen såkalte *where-clauses* for å angi parameterskranker, i stedet for å bruke allerede definerte klasser og interfacer. En *where-clause* er rett og slett en skranke på en klasseparameter som bestemmer at klassen som sendes med som aktuellparameter, må ha implementert visse metoder. Det er ikke lenger krav til at klassepara-

meteren skal være en subklasse av en annen spesifisert klasse/interface, som blant annet må være synlig fra både definisjons- og brukssted.

I tillegg har de valgt å erstatte spiss- med hakeparenteser siden tabeller (arrayer) på en måte er parametriserte klasser, og disse allerede bruker dette. En enkel bruk av den nye formen for skranke kan illustreres med følgende eksempel:

```
interface SortedList[T] where T { boolean lt(T t); } {
    ...
}
```

I koden over er det et krav at klassen `T` som sendes med som parameter, må ha implementert metoden `lt` med den riktige typen på parameteren. Ved å benytte where-clauses, vil det være enklere å benytte allerede definerte klasser som aktuelle parametere. Siden det kun er krav til hvilke metoder parameterklassen skal ha, vil det ikke lenger være behov for at klassen (eller interfacet) er kjent både ved definisjonen av den generiske klassen og ved definisjonen av den aktuelle typeparameteren (som kan være langt unna brukstedet).

Forfatterne hevder at blant annet [8] sin mekanisme blir problematisk særlig i større systemer hvor klasser etter hvert må implementere mange forskjellige interfacer. Selv om klassen inneholder de nødvendige metodene, vil ikke dette være nok.

Denne tankegangen er jeg enig i, og jeg har i jobbsammenheng selv erfart hvor hensiktsmessig en mekanisme med where-clauses ville vært; særlig når man jobber mye med allerede eksisterende klasser som man ikke selv har definert.

I denne mekanismen er det heller ikke definert noe subtypeforhold mellom `List[Person]` og `List[Object]` selv om `Person` er en subklasse av `Object`. Med andre ord blir ikke klassene satt i et hierarki etter samme kovariante tankegang som Thorup mener er en viktig egenskap ved sine virtuelle typer [10].

Denne mekanismen støtter også bruk av primitive typer på samme måte som Agesen [8], men i tillegg kan man bruke dem som parametere selv om det er definert where-clauses som setter krav til metoder.

Utgangspunktet for å få til dette, er at de primitive typene har fått tilegnet metodenavn for de operasjonene som primitivene støtter. Blant annet tilsvarer operasjonen ”mindre enn” (<) metoden `lt`, mens ”er lik” (==) tilsvarer `equals`. På denne måten kan man sende inn typen `int` der det kreves at klassen inneholder metoden `equals`.

Man kan stusse litt på valget av metodenavnene. For eksempel bruker de det fulle navnet `equals`, mens man samtidig har valgt det forkortede navnet `lt` i stedet for `lessThan`.

Generelt i denne mekanismen kan det bli problematisk hvis man skal som parametere, gi med klasser man selv ikke har definert. Disse kan kanskje ha ”feil” metodenavn selv om metoder har den korrekte funksjonaliteten. Dette kan skape et behov for navneendring av metoder. I slutten av dette kapittelet kommer jeg nærmere inn på dette.

3.3.2 Generell type, List

I de neste avsnittene skal jeg presentere hvordan de forskjellige problemstillingene kan løses ved hjelp av parametriserte typer som benytter where-clauses. En enkleste klasse `List` har ingen skranker tilknyttet parameteren, slik at koden her blir helt lik den som stod i kapittel 2; bortsett fra at man må benytte hakeparenteser.

```
class List[A] {
    void put(A elem) {...}
    A get(int pos) {...}
}

...

List[Person] l = new List[Person]();
```

3.3.3 Forutsatt metode, HashTable

I eksempelet rundt `HashTable`, trenger man ikke lenger å operere med interfacet `Hashable`. Siden man nå benytter where-clauses, kan man bestemme at klassen som skal legges inn i tabellen, må inneholde den etterspurte metoden `hashCode`. Ved bruk av where-clauses, vil biblioteket kunne implementeres som følger:

```
package hashing;

class HashTable[Key, Value] where Key { int hashCode(); } {
    Value values[];

    void put(Key key, Value value) {
        array[key.hashCode()] = value;
    }

    Value get(Hashable key) {...}
}
```

Brukerapplikasjonen kan være implementert som følger.

```
import hashing;

class Person {
    int hashCode() {...}
}

...
```

```
HashTable[Person, int] h = new HashTable[Person, int]();
```

Hvis man sammenligner koden over med den som Agesens forslag tok utgangspunkt i, ser man at koden blir mindre og mer kompakt når man slipper å definere forskjellige interfacer.

3.3.4 Forutsatt metode med gitt typeparameter, Prioritetskø

Ved bruk av where-clauses, får man også lov å la de generiske klasseparameterne benyttes videre ved definering av grensesnittet til de påkrevde metodene. Med andre ord er det ikke noe problem å sette krav til binærmetoder. Slike skranker kan illustreres med eksempelet rundt prioritetskøer.

```
package priorityQueue;

class PriorityQueue[T]
    where T { boolean lessThan(T); } {
    T queue[];

    void insert(T elem) {
        if (elem.lessThan(queue[i]))...
        ...
    }
}
```

Biblioteket kan videre benyttes i en applikasjon som følger:

```
import priorityQueue;

class Prime {
    boolean lessThan(Prime other) {...}
}

PriorityQueue[Prime] pq = new PriorityQueue[Prime]();
pq.insert(new Prime(5));
```

3.3.5 Utvidelse av en klasse, Mix-In

I artikkelen poengterer forfatterne spesifikt at de støtter deklarasjoner som sier at den generiske klassen er en subklasse av en av aktuellparameterne. Det vil dermed ikke være noe problem å skrive den generelle klassen `Relations`.

```
package relations;

class Relations[C]
    where { boolean lessThan(C);
            boolean equals(C); }
    extends C {
    boolean greaterThan(Relations[C] other) {
        return (other.lessThan(this) && notEqual(other));
    }
```

```
    boolean notEqual(Relations[C] other) {
        return !other.equal(this);
    }
}
```

Brukerprogrammet blir relativt likt som med mekanismen til Agesen [8], men ved definisjon av den aktuelle parameterklassen, trenger man ikke lenger å ha tilgang til et interface som også må være synlig fra definisjonsstedet til den generiske klassen:

```
import relations;

class SimpleRelations {
    boolean lessThan(SimpleRelations other) {...}
    boolean equal(SimpleRelations other) {...}
}

Relations[SimpleRelations] advancedRelations =
    new Relations[SimpleRelations];
```

Igjen kan man se at koden blir noe mer kompakt, og i mine øyne også mer lettlest. Bruk av where-clauses klarer å uttrykke på en stilren måte hvilke metoder som er påkrevd, samtidig som man klarere skiller ut hvilken klasse som blir en subklasse av hvilken.

En ulempe ved bruk av where-clauses, er at spesifikasjoner kan bli ganske lange. Hvis man har krav om at en klasse må ha implementert en mengde metoder, kan dette være et tegn på at man egentlig ønsker å uttrykke at en aktuellparameter skal være en subklasse av en gitt klasse. I slike tilfeller vil det kanskje være mest hensiktsmessig å kunne beskranke ved hjelp av implements og/eller extends. Det beste hadde kanskje vært å ha mulighet til å kunne beskranke på begge måter.

3.3.6 Implementasjon av mekanismen

Andrew C. Myers og kollegene har tatt utgangspunkt i at deres parametriserte klasser skal støtte separatkompilering, og dette får de også til ved hjelp av skrankene som settes på parameterne. Ved å ta utgangspunkt i skrankene, kan man statisk sjekke de generiske klassene selv om man ikke vet hvilke parametere de skal ha.

Som jeg nevnte var et problem for Agesen [8], kan det for denne mekanismen også oppstå flertydighet ved spesielle instansieringer av en parametrisert klasse. En generisk klasse kan få to metoder med samme grensesnitt ved enkelte instansieringer. Strategien som foreslås i forbindelse med where-clauses, er å la kompilatoren velge den metoden som passer ”best”, men den gir feilmelding hvis den ikke klarer å finne en unik ”beste”.

Denne generiske mekanismen er foreslått innført med en homogen implementasjon, og en generisk klasse blir dermed kompilert ned til én kodefil i stedet for én for hver bruk. Dette gjør at de unngår at koden for en klasse blir gjen-

tatt flere ganger, noe de selv kaller en *"code blowup"*. Siden de kun opererer med én klassefil, må de i tillegg til selve koden lagre informasjon om de forskjellige skrankene.

I utgangspunktet ønsket forfatterne, så lenge det var hensiktsmessig, å spare plass ved å benytte en homogen implementasjon. Dette valget medfører som de selv sier, at man kan jobbe med én kompilert klassefil (selv etter at den er hentet inn av loaderen i JVM-en), men samtidig får de en vanskeligere oppgave med å integrere mekanismen i nåværende Java.

Når et Java-program startes i implementasjonen til [15], opprettes det ett objekt for hver parametriserte klasse, og dette inneholder blant annet en peker til den del av minnet (*constant pool*) hvor den generelle informasjonen om klassen er lagret. Siden en parametrisert klasse kan instansieres med forskjellige typer, vil det under kjøring være visse forskjeller fra instans til instans.

Dette har man løst ved at hver instans av en parametrisert klasse, i tillegg får tilegnet ett spesielt instansobjekt hvor klassen kan lagre det som er spesielt for denne instansen. På denne måten kan man lagre så mye felles som mulig, samtidig som man har mulighet til å separat lagre den informasjonen som er spesifikk fra bruk til bruk.

Selv om [15] benytter en homogen implementasjon, støtter de likevel bruk av primitive typer. Siden nesten alle primitive typer tar like stor plass i minnet som det en objektreferanse gjør, er det ikke noe vanskeligere å beregne relativadresser for den "nye" kompilerte koden. I tillegg har man ved hjelp av det instansspesifikke kjøreobjektet, løst andre problemer som er tilknyttet primitivene.

Likevel er ikke denne metoden fullgod. Som jeg sa, tar *nesten* alle primitive typer like stor plass i minnet som en objektreferanse, men dette er ikke tilfellet for `double` og `long`. Derfor behandles disse tilfellene spesielt, og dette løser de faktisk ved å generere egne klassefiler i de tilfeller der man benytter en av disse primitivene som aktuellparameter. Med andre ord er ikke metoden fullt homogen, men så godt som det har vært mulig etter deres filosofi.

Forfatterne har utviklet en modifisert kompilator og kodeinterpreter (JVM) som støtter deres nye mekanisme, og de har hatt positive tester både når det gjelder størrelse på koden og ikke minst ytelse. Ved å fjerne all bruk av castinger, har de klart å øke effektiviteten under programkjøring, på tross av at en homogen implementasjon i utgangspunktet normalt vil redusere hastigheten noe. Blant annet sier de selv at en generisk klasse `List` kjører 17% raskere på deres prototyp enn den tilsvarende klassen skrevet ut i fra det generiske idiom.

Dette tallet bør nok leses med den viten at andre med tilsvarende mekanismer, ikke har fått fullt så gode resultater. Et slikt resultat vil jo være svært avhengig av testprogrammet, blant annet av hvor ofte det utføres operasjoner som ved det generiske idiom måtte ha casting.

3.4 Parametriserte typer, Bracha mfl.

Relativt tidlig i diskusjonen rundt innføring av generiske typer i Java, ble det lansert et forslag til utvidelse, kalt Pizza [18]. Pizza ligner en del på Agesens forslag til generiske typer (med blant annet bruk av implements og extends for å spesifisere de generiske parameterne), men hadde en del andre svakheter som forfatterne senere rettet opp i sitt neste forslag til utvidelse kalt GJ (Generic Java). Denne er beskrevet i artikkelen *"Making the future safe for the past: Adding Genericity to the Java Programming Language"* [6].

GJ innfører en fullverdig generisk syntaks, omtrent som i [8], men koden blir deretter oversatt til helt standard Java før kompilering, og man behøver dermed ikke å gjøre noe med verken loaderen eller linkeren i JVM-en. Den generiske koden blir oversatt til kode som benytter det generiske idiom, og kan blant annet derfor ikke brukes med primitive typer.

3.4.1 Generelt om den generiske mekanismen i GJ

Med GJ innfører forfatterne et design som utvider Java-språket med generiske typer og metoder. I motsetning til de andre mekanismene jeg har gått gjennom, støtter [6] også generiske parametere brukt på enkeltestående metoder; noe som kan være nyttig hvis man for eksempel har en klasse som inneholder uavhengige "verktøy-metoder" som hver kan ha spesifisert sine egne generiske typer, gjerne statiske metoder.

Men man kan også angi parametere på klassenivå, og slike klasseparametere er tilgjengelige i hele klassen bortsett fra i de statiske attributtene. Med den implementasjonsmetoden det er lagt opp til (som er homogen), vil de statiske attributtene bli delt mellom alle de forskjellige instansene, selv de med ulike generiske typer. Dermed blir det umulig at de generiske klasseparameterne også skal gjelde for dem.

Men man kan altså likevel definere egne generiske parametere for de statiske metodene, siden GJ støtter generiske parametere på metodenivå. Når man skal angi generiske parametere på en metode, må de angis før selve typen og navnet til metoden.

Eksempelvis kan man tenke seg en klasse som har en statisk metode som tar en vektor som parameter og som returnerer objektet i vektoren med størst verdi, definert ut i fra et interface `Comparable` med en metode `compareTo`. Klassen `VectorMaxElement` kan skisseres som følger:

```
static class VectorMaxElement {
    static <A implements Comparable<A>> A max(Vector<A>) {
        ...
    }
}
```

Som jeg nevnte, implementeres denne mekanismen tett opp mot tanken bak det generiske idiom. Ved kompilering blir generiske parametere fjernet og

man står igjen med såkalte *rå typer*. Typen `List<A>` vil bli endret til `List`. Variable inne i klassen typet med en parametertype, får tilegnet den mest generelle typen som samtidig oppfyller beskrankningen av parameteren. Hvis en generisk parameter ikke er beskranket, vil typen dermed bli byttet ut med den mest generelle typen `Object`. For at typene i den endrede koden skal tilsvare den opprinnelige, legges det inn castinger der det er nødvendig.

I tillegg vil det under translasjonen bli lagt inn såkalte bro-metoder for at redefinering (overriding) av metoder skal bli korrekt etter at de generiske typene er ersattet med vanlige Java-typer. Under har jeg satt opp et interface og en klasse som viser hvordan dette kan bli seende ut i GJ. I eksempelet returnerer `compareTo` heltallsdifferansen mellom parameteren og objektet selv.

```
interface Comparable<A> {
    int compareTo(A other);
}

class Byte implements Comparable<Byte> {
    byte value;

    int compareTo(Byte other) {
        return value - other.value;
    }
}
```

Etter reglene i GJ skal her `compareTo` i `Byte` redefinere (override) den i `Comparable`. I oversettelsen vil imidlertid `compareTo` i `Comparable` bli typet om til klassen `Object`, mens den i klassen `Byte` vil beholde `Byte`-typen. Derfor må det settes inn en egen ”brometode” som gjør at klassen `Byte` oppfyller kravene fra interfacet `Comparable`. Oversettelsen blir som følger:

```
interface Comparable {
    int compareTo(Object other);
}

class Byte implements Comparable {
    byte value;

    int compareTo(Byte other) {
        return value - other.value;
    }

    // brometode som redefinerer compareTo i Compareble.
    int compareTo(Object other) {
        return compareTo((Byte)other);
    }
}
```

Filosofien som ligger i bunn for mekanismen til Bracha og kollegene, legger i utgangspunktet opp til en homogen implementasjon. Man oversetter rett og slett generisk kode til vanlige Java-filer med den generelle typen `Object`

eller andre generelle typer. Ved å gjøre det på denne måten, blir det enklere å integrere mekanismen i nåværende Java, men på den annen side, mister man en del av det en generisk mekanisme kan tilby.

Siden GJ ved kompilering fjerner en del informasjon rundt typer i en separatkompilert klassefil, vil det i utgangspunktet være vanskelig å få til kompilering av brukerprogrammer bare ved tilgang til klassefilene til de kompilerte generiske klassene. Dette har man løst ved å inkludere noe av typeinformasjonen i et signaturfelt i byte-koden. Dette feltet er i utgangspunktet beregnet som et generelt kommentarfelt, og overses av vanlige JVM-er. På denne måten kan gamle JVM-er fortsatt klare og lese de kompilerte filene.

Kompilerte generiske klasser, blir altså gjort om til ”vanlige” Java-klassefiler, men hvor det er lagt inn litt tilleggsinformasjon. Dette åpner muligheten for noe forfatterne har kalt *retrofitting*. Retrofitting er en mekanisme som skal gjøre det mulig å lage en generisk versjon av en klassefil som er skrevet ved hjelp av det generiske idiom.

Tanken er at man har en vanlig, ferdigkompilert bibliotekskласse som er skrevet ved hjelp av det generiske idiom, og hvor grensesnittet er kjent. Ved kun å skrive en generisk signaturfil som tilsvarer grensesnittet til den eksisterende klassen, kan man bruke klassen som om den skulle vært skrevet med ”ekte” generiske typer. Man kan altså i ettertid ”legge til” den generiske biten ved å samkjøre en eksisterende klassefil med et nytt generisk grensesnitt.

Dette gjør det ganske enkelt å konvertere gamle bibliotekskласser til å støtte de generiske typene innført av GJ. Man trenger ikke å skrive om bibliotekene, men kun å tilordne tilleggsfilene beregnet på retrofitting.

Siden gamle JVM-er kan lese nye klassefiler, samtidig som gamle klassefiler skrevet med det generiske idiom kan gjøres ”generiske” ved et tillegg, reklamerer forfatterne med at deres implementasjonen på en måte *både* blir framover- og bakoverkompatibel, i motsetning til de andre mekanismene som i hovedsak kun vektlegger at nye programmer skal kunne kjøres på vanlige JVM-er.

En slik løsning kunne man sikkert ønske seg for de andre mekanismene også, men for dem vil det i utgangspunktet være mye vanskeligere. Disse jobber ikke like nært opp til det generiske idiom, og det er nettopp dette som gjør at det er mulig for GJ.

GJ støtter heller ikke den kovariante tankegangen til Thorup [10]. Med andre ord vil ikke `List<Person>` være en subtype av `List<Object>`. Dette valget begrunner de med at det ellers vil være umulig å statisk sjekke koden. For hvis klassene skulle ha et subtypeforhold, kan man se for seg følgende problematiske eksempel:

```
class C {
    List<Integer> l1 = new List<Integer>;
    List<Object> l2 = l1; // dette ville være lov.

    l2.add(new Person()); // dette vil føre til en feil,
```

```
                                    // men kan ikke sjekkes statisk
}
```

Tilsvarende problem som jeg har skissert over, gjelder også for blant annet Agesens forslag og forekommer også ved bruk av vanlige tabeller (arrayer) i Java. Ved alle tilordninger til tabeller utføres det derfor en typesjekk under programkjøring. GJ ønsker at kode skal statisk sjekkes så fremt det er mulig, og forfatterne har derfor bestemt at instanser av generiske klasser ikke skal ha noe subtypeforhold.

3.4.2 Styrke i forhold til eksempler

Syntaksen og mulighetene som de generiske parameterne gir, er i utgangspunktet veldig like de Agesen og kollegene fra [8] tilbyr. GJ tilbyr samme muligheter å sette skranker på generiske parametere med nøkkelordene *implements* og *extends*. I tillegg tilbyr de generiske parametere på metodenivå, noe som er spesielt interessant for statiske metoder.

I GJ får man derfor uttrykt de samme konstruksjoner som med Agesen sin mekanisme, men under kompilering er de to mekanismene veldig forskjellige. GJ oversetter som sagt et program til vanlig Java, mens [8] beholder de generiske typene mer som de er. GJ får av denne grunn en rekke begrensinger.

3.4.3 Begrensninger med mekanismen

GJ tilbyr for så vidt en generisk mekanisme, men fokuset under utformingen har vært svært mye på kompatibilitet både ”forover” og ”bakover”, samtidig som de insisterer på en homogen implementasjon. Resultatet blir dermed noe ”ad hoc”, hvor det må legges inn en del kunstige konstruksjoner samtidig som en del vil bli ulovlig.

Dette stikker blant annet i at GJ under oversettelsen, fjerner de generiske typene og man står igjen med såkalte *rå typer*. Under programkjøring vil derfor de generiske typene være ukjente, og som en følge av dette, oppstår det noen begrensninger og problemer. Det vil blant annet være ulovlig å instansiere objekter ut i fra en generisk parameter. Dette vil være umulig siden den aktuelle typen ikke vil være kjent ved utførelsen. Kommandoer som “`refA = new A();`” der `A` er en generisk parameter, vil blant annet ikke være lovlige.

Jeg nevnte også tidligere at GJ under oversettelsen er nødt til å sette inn såkalte brometoder for at metodene skulle oppfylle kravet om redefinisjon i Java. Hvis en slik brometode ikke har noen parameter, men kun en returverdi, vil det også oppstå problemer ved oversettelsen. I en klasse `Iterator`, vil det som regel være en slik metode `next`.

```
class Interval implements Iterator<Integer> {
    ...
    Integer next() {...}
    ...
}
```

Denne klassen vil bli oversatt til følgende klasse som inkluderer en brometode som skaper problemer.

```
class Interval implements Iterator {
    ...
    Integer next() {...}

    Object next() { // brometode
        return next();
    }
    ...
}
```

Dette er ikke lovlig Java-kildekode siden `next` i de to versjonene ikke kan bli skilt fra hverandre fordi de har like argumenter. Man kan heller ikke bare fjerne brometodene da regler for redefinisjon ikke vil bli overholdt. Heldigvis kan dette løses på byte-kodenivå, fordi man der også ser på resultattypen når man skal skille metoder. Slike konstruksjoner må derfor oversettes direkte til byte-kode, siden en eventuell Java-kildekode ikke vil kunne kompileres.

Til slutt vil jeg ta opp et annet problem som oppstår, og som flere av de andre artiklene tar opp under omtale av GJ. Dette omhandler et eksempel rundt krypterte kanaler. Man kan tenke seg at man i en klasse ønsker seg en liste med kun krypterte kanaler.

```
class SecureChannel extends Channel {
    ...
}

class Security {
    LinkedList<SecureChannel> secureChannels;
    ...
}
```

Når et slikt program kompileres i GJ, gjøres `secureChannels` om slik at den er av typen `LinkedList`. Med andre ord vil det være mulig å legge inn kanaler som ikke er krypterte, og disse kan igjen brukes for å tappe `Security` for informasjon. Hvis hele programmet var skrevet i GJ, ville ikke problemet være like relevant grunnet spesifikt innlagte sjekker, men deler av et programmer kan skrives i vanlig Java eller direkte i bytekode, og da vil det være relativt enkelt å omgå sikkerhetsrutinene på denne måten.

Forfatterne av GJ prøver å avdramatisere dette ved å foreslå en enkel løsning på problemet. Hvis man oppretter en spesifikk subklasse av `LinkedList`, vil typene beholdes selv etter kompilering.

```
class SecureChannelList extends LinkedList<SecureChannel> {
}

class Security {
    SecureChannelList secureChannels;
    ...
}
```

I dette tilfellet vil problemet være løst siden typen `SecureChannelList` vil beholdes hele veien. Likevel er det mange, inkludert meg, som ikke liker denne løsningen. Her er det opp til programmereren å kjenne til problemet og omgå det selv. Med andre ord vil det trolig være en del som går i samme felle og tilbyr programmer med relativt store sikkerhetshull.

Innledningsvis fortalte jeg at GJ er en videreføring av et tidligere prosjekt under navnet Pizza [18]. Delvis parallelt med GJ har det også blitt utviklet et litt større og kraftigere språk med navn NextGen [17]. Språket NextGen er et "supersett" til GJ, i den betydning at alle lovlige GJ-konstruksjoner er lovlig i NextGen, men ikke motsatt.

NextGen løser flere av problemene som jeg her har skissert for GJ, men språket har til gjengjeld en mer kompleks implementasjon hvor man blant annet lagrer mer informasjon om typer. Likevel bygger mekanismen på det generiske idiom, og dette forhindrer muligheten for generering av virkelig rask kode.

3.5 Sammendrag av de generiske mekanismene

Siden artiklene som er diskutert over, delvis er skrevet uavhengig av hverandre og dermed har litt forskjellig fokus, er det ikke så enkelt å sammenligne dem, og i hvert fall ikke trekke bastante konklusjoner. Jeg vil likevel i dette avsnittet prøve å sammenligne og diskutere sider ved de foreslåtte mekanismene, og etter hvert også trekke noen konklusjoner over hvilke sider ved hver mekanisme som er best.

	Thorup	**Agesen mfl.**	**Myers mfl.**	**Bracha mfl.**
List	✓	✓	✓	✓
HashTable	✓	✓	✓	✓
Prioritetskø	✓	✓	✓	✓
Mix-in		✓	✓	✓
Separatkomp.	✓	✓	✓	✓
Homogen	✓		✓	✓
Kovarians	✓			
Primitive typer		✓	✓	

Figur 3.2 Sammendrag av de forskjellige generiske mekanismene.

Ut i fra disse sammenligningene vil jeg prøve å skissere én "samlet" mekanisme hvor jeg trekker inn det beste fra de forskjellige artiklene. Denne me-

kanismen, sammen med en annen jeg skal presentere i kapittel 4, vil jeg ta utgangspunkt i ved implementasjon av design-patternene i kapittel 5.

3.5.1 Virtuelle typer kontra parametrisering

I utgangspunktet er det Thorup sine virtuelle typer som skiller seg mest ut i fra de andre, og den største svakheten til Thorup, er at mekanismen ikke vil kunne være bli like kraftig som de parametriserte typene (som vi for eksempel så med Mix-in). Fordelen er at han slipper å innføre et ekstra begrep ”generisk” klasse (eller meta-klasse). Bruk av generiske klasser vil blant annet gjøre det vanskeligere å definere et klart klassehierarki, men jeg mener at dette er et svakere argument enn at man ikke klarer å få skrevet klasser som `Relations` fra Mix-in.

Man kan på den annen side, som en del nyere artikler beskriver, vurdere å kombinere teknikkene med parametrisering og virtuelle typer, og dette er blant annet prøvd ut i artikkelen *”Unifying Genericity”* [22], som også er skrevet av Thorup, nå sammen med Mads Torgersen. I denne artikkelen opererer man med såkalte *strukturelle virtuelle typer*, som tar utgangspunkt i de virtuelle typene jeg presenterte, men hvor man samtidig også tilbyr parametrisering.

Jeg har likevel valgt å ikke fokusere på denne kombinerte mekanismen, blant annet fordi parametriserte typer passer bedre inn i Java sin språkfilosofi, samtidig som dette er en kraftig nok mekanisme i seg selv. Jeg har derfor heller valgt å komme nærmere inn på en helt annen mekanisme som kan styrke Java på litt andre måter. Denne kommer jeg tilbake til i neste kapittel.

I et språk som Beta, som bygger mye av sin struktur rundt virtuelle typer, vil nok [22] være en mekanisme som kanskje er bedre egnet. Dette har blant annet Bjørn Willassen, som jeg har jobbet noe parallelt med, sett mer på i sin hovedfagsoppgave *”Generiske Språkbegreper & Design-Mønstre”* [23].

3.5.2 Kovarians

For hver av mekanismene jeg har presentert i dette kapittelet, har jeg diskutert hvorvidt de har implementert kovariant typing eller ikke. På den ene siden får man ved kovariant typing et ”rikere” klassehierarki, men på den annen side er man nødt til å legge inn runtime-tester da koden ikke lenger er statisk validerbar (se slutten av kapittel 3.4.1).

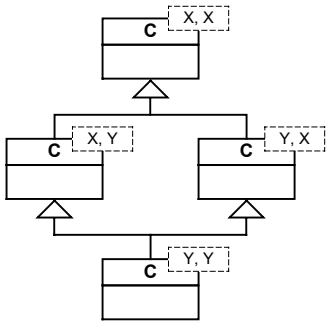
Thorup vektlegger at kovarians er viktig for hans mekanisme, og han kritiserer flere av de andre forslagene siden parametriserte typer i utgangspunktet ikke legger opp til dette på samme måte. Likevel kommer han ikke med noe spesielt godt argument for hvorfor det er viktig. Jeg vil derfor nå diskutere litt fordeler og ulemper rundt kovarians.

Som jeg allerede har vært inne på, er tabeller (arrayer) i Java allerede kovariante, og dette kan være behagelig når man programmerer ved hjelp av det generiske idiom. I generelle klasser som skal kunne fungere med forskjellig

type objekter, vil man ofte måtte operere med `Object[]`-pekere. I brukerprogrammer vil man derimot ofte jobbe med ”faktiske” tabeller som består av én type objekter, og disse vil dermed være typet med for eksempel `Person[]`-pekere. I slike situasjoner er det viktig at en `Object[]`-peker kan peke på en tabell av personer.

Ut i fra samme tankegang kan man kanskje motivere for samme behov ved generiske mekanismer: altså at man har en generisk klasse `List<X>` i den generelle koden, samtidig som man der også er nødt til å operere med enkelte pekere til `List<Object>`. I et brukerprogram hvor man for eksempel jobber med klasseinstansen `List<Person>`, vil det da på tilsvarende måte, kunne være et behov for at en `List<Object>`-peker må kunne peke på et `List<Person>`-objekt.

Av denne grunn kan det med andre ord være et behov for slik kovarians også ved bruk av generiske mekanismer. Det er imidlertid et viktig poeng her at man med en god generisk mekanisme, burde kunne få forandret alle pekerne av `List<Object>` til `List<Person>`, slik at man likevel ikke får så stort behov for kovarians. I neste kapittel vil jeg presentere en annen generisk tankegang hvor man får løst mye av denne problematikken.



Figur 3.3 Eksempel på ”multippelt” instanshierarki ved kovarians.

Kovarians er nok også en mekanisme som faller mer naturlig ved bruk av virtuelle typer, siden man der kun jobber med vanlige klasser og ikke har noe eget begrep ”generisk klasse”. Likevel *kan* man for parametriserte typer også innføre kovarians, men det må i så fall defineres klare regler for hvor forskjellige klasseinstanser skal plasseres i klassehierarkiet. Blant annet kan en klasse ha flere generiske parametere, noe som kan skape et hierarki hvor man får noe *lignende* multippel arv. Se Figur 3.3.

Gjennom hele denne oppgaven vektlegges det at det er ønskelig å statisk kunne sjekke kode, både den i de separate generiske klassene, og den der disse brukes. Siden behovet for kovarians reduseres med en god generisk mekanisme, samtidig som kovarians kan svekke hastigheten på programeksekveringen, har jeg valgt å ikke vektlegge dette punktet tyngst i valget blant de fire mekanismene.

Hvis det likevel skulle vise seg å være et behov for kovarians, bør det være relativt enkelt å innføre det for parametriserte typer også, men det må da som sagt samtidig legges inn enkelte runtime-tester.

3.5.3 Beskrankning av parametere

Hvis man ser bort fra virtuelle typer og kun konsentrerer seg om de parametriserte mekanismene, kan man velge mellom to måter å beskranke parametere på. Agesen og Bracha benytter nøkkelordene extends og implements, mens Myers bruker where-clauses. Dette er løsninger som begge er gode på hver sine måter.

I mange tilfeller vil man kun trenge å beskranke med tanke på enkelte metoder, mens man i andre klasser kanskje må sette krav til at en aktuell parameter skal være en subtype av en annen gitt type. Den første formen skaper større uavhengighet mellom definisjonen av de generiske klassene og av de aktuelle parameterklassene, mens den siste blant annet vil være aktuell for flere design-patterns hvor et pattern kan bestå av flere samspillende klasser.

Det er heller ikke noen grunn til at den ene formen for beskrankning skal forhindre bruk av den andre. Jeg vil derfor i den generiske mekanismen som jeg skal bruke videre, tillate *både* implements, extends og where-clauses. Om man forsøker å lage en mest mulig enhetlig syntaks, burde ikke dette bli forvirrende for brukeren, og implementasjonsmessig bør det heller ikke være veldig belastende for kompilatoren. Antakeligvis kan man bruke en homogen eller en heterogen implementasjon alt etter om man vektlegger plass eller tid

3.5.4 Generelt

Som jeg var inne på i kapittelet hvor jeg presenterte GJ [15], bærer denne mekanismen preg av å være litt for integrert med det generiske idiom og nåværende JVM-er. Siden dette går ut over mulighetene i mekanismen, vil jeg heller si at Agesen og Myers sine forslag er bedre; hvorav Agesen opererer med en heterogen og Myers med en homogen implementasjon.

Siden jeg i denne oppgaven ikke skal fokusere for mye på den konkrete implementasjonen, blir de på en måte ganske sidestilte da syntaksen i forslagene er relativt like. Videre i denne oppgaven har jeg derfor valgt å ta utgangspunkt i en mekanisme lik Myers og kollegene sin, men hvor jeg i tillegg tillater å beskranke parameterne med nøkkelordene extends og implements på tilsvarende måte som Agesen gjør.

Det siste jeg vil nevne i dette sammendraget, er at man i GJ diskuterer en mekanisme som går ut på at hvis man *ikke* sender med en aktuellparameter til en parametrisert klasse, skal den instansieres med `Object`. Ved å gjøre det, fjerner man på en måte begrepet generisk klasse, men på den annen side er ikke dette en fullstendig løsning på kompatibilitetsproblemet. Denne tankegangen vil for eksempel ikke fungere hvis man har satt beskrankninger med et inter-

face på en parameter. Det vil da ikke lenger være én klasse som naturlig peker seg ut til å være ”standard” aktuellparameter.

3.6 Mulighet for navneendring og alias

Til sist i dette kapittelet, vil jeg her komme inn på et forslag til løsning på et problem jeg selv har erfart når jeg har jobbet med parametriserte klasser.

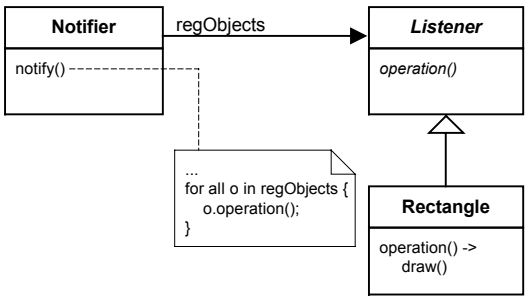
Ved bruk av generiske klasser og biblioteker, vil man av og til komme opp i situasjoner hvor det kan være ønskelig å ha mulighet til å endre navn på metoder i subklasser. Parametrisering kan også fremkalle unødvendig tungskrevne betegnelser for klasser, og her hadde det derfor vært greit å ha en mekanisme for å definere alias for visse klasseinstanser.

3.6.1 Navneendring

Et av målene ved generisk programmering, er at det skal være enklere å kunne skrive generelle klasser hvor mer kode ligger i selve biblioteket, samtidig som det skal virke som at koden er ”skrevet for stedet” på hvert av de bruksteder den hentes inn.

Et av problemene som kan oppstå når metoder som skal kunne benyttes i forskjellige sammenhenger puttes inn i én generell klasse, er at metodenavnet må være veldig abstrakt. I enkelte tilfeller kan metoder ende opp med navnet `operation`, et navn som er noe ”mangelfullt” når metoden benyttes i en konkret sammenheng.

Eksempelvis kan man se på et bibliotek som består av to klasser, hvor en klasse `Notifier` inneholder en liste med objekter av klassen `Listener`. `Notifier` inneholder også en metode `notify` som trigger et metodekall i alle objekter som er registrert i denne lista.



Figur 3.4 Eksempel med navneendring.

Klassen `Notifier` er tenkt å ligge i et bibliotek, så metodenavnet som skal trigges, må bli bestemt når biblioteket skrives. Biblioteket skal kunne brukes i forskjellige sammenhenger, og et av de mest generelle metodenavnene som finnes er da `operation`. Subklasser som i ettertid skrives, vil sjelden ønske å

ha slike metodenavn, så det melder seg derfor et behov for å kunne angi at en redefinisjon av en metode i en subklasse skal ha et annet navn.

I Figur 3.4 er det illustrert et eksempel hvor en metode `draw` i en klasse `Rectangle`, skal være en redefinisjon av metoden `operation` i `Listener`. UML har ikke definert noen måte å uttrykke dette på, så jeg har derfor foreslått en midlertidig syntaks hvor jeg benytter symbolet `->` for å angi navneendring (se Figur 3.4).

I en klassedeklarasjon vil det da være behov for en syntaks som gjør det mulig å gi beskjed om at en metode `operation` i en subklasse skal ha navnet `draw`. Dette kan for eksempel gjøres ved følgende kode:

```
class Rectangle extends Listener <where operation -> draw> {
    ...
}
```

Denne syntaksen kan være egnet hvis man selv definerer subklassen, men den holder ikke hvis man skal bruke ovennevnte bibliotek med ferdig definerte klasser. I disse tilfellene vil det være behov for å kunne uttrykke navneendringen ved enhver bruk av klassen. I så fall er man nødt til å gjenta hele definisjonen hver gang typen nevnes; noe som kan skape et behov for bruk av alias (se neste avsnitt). Man kan tenke seg følgende syntaks for å uttrykke dette uten noen alias-mekanisme:

```
Rectangle<where operation -> draw> refRectangle;
refRectangle = new Rectangle<where operation -> draw>();
```

Her er ikke syntaksen spesielt god, men det viser i hvert fall hva jeg kunne ønske at man fikk uttrykt i et språk. Det vil her selvfølgelig bli mange problemer som måtte avklares før man hadde et konsistent system. Er for eksempel én klasse med to forskjellige navneendringer fremdeles samme klasse? Jeg går ikke nærmere inn på dette, da jeg mener det rimelig greit bør kunne finnes konsistente løsninger.

3.6.2 Alias

Når man programmerer med parametriserte klasser, kan det raskt oppstå klassekonstruksjoner som er tunge å skrive hele veien i et program. Særlig når en klasse tar flere generiske parametere og eventuelt også har med navneendring, vil et program både bli mer lettlest og enklere å skrive hvis man kunne definere at midlertidig navn på en spesiell instans av en parametrisert klasse.

I utgangspunktet kan man for eksempel tenke seg følgende kode som benytter en parametrisert klasse:

```
Stack<int, Person> stack = new Stack<int, Person>();
...
if (stack instanceof Stack<int, Person>) {...}
```

Tanken med den nye syntaksen er at man skal kunne sette et alias på en spesiell instans av en parametrisert klasse, og koden over kan dermed erstattes med kode som er enklere og mer lettlest.

```
alias PersonStack as Stack<int, Person>;
PersonStack stack = new PersonStack();
...
if (stack instanceof PersonStack) {...}
```

Man kan implementere dette med en preprosessor som tekstlig erstatter det man ønsker, men for å være en fullgod løsning, bør nok en slik mekanisme legges inn i selve kompilatoren. På denne måten får man i tillegg mulighet for å utføre enkle semantiske sjekker.

Ved å innføre nøkkelordet `alias`, oppnår man blant annet følgende:

- Man slipper å gjenta generiske parametere og eventuelt navneendringer hele veien gjennom programmet.
- Koden blir mer lettlest.
- Man kan definere skopregler for aliaset, på samme måte som andre attributter. I utgangspunktet bør et alias, som de fleste andre definisjoner, være gjeldende innenfor en programblokk.

I motsetning til hvis man hadde løst substitueringen med en preprosessor, kan man nå legge inn litt intelligens for hvor man skal utføre en erstatning og hvor man ikke skal det. En makrobasert, tekstlig løsning vil typisk erstatte alle forekomster, selv om det kanskje bare er ønskelig innenfor en programblokk.

Siden kompilatoren substituerer ut aliasene før selve kompileringen, vil man kunne betrakte et alias som samme klasse som en tilsvarende generisk klasse med like aktuellparametere. Følgende tilordning vil derfor være fullt lovlig:

```
alias PersonStack as Stack<int, Person>;
...
Stack<int, Person> stack = new PersonStack();
```

Bruk av alias og navneendring vil være spesielt nyttig ved parametriserte klasser som jeg har omtalt i dette kapittelet. I neste kapittel skal jeg se nærmere på en annen form for generisk tankegang, og i denne sammenheng vil behovet for alias og navneendring være mindre.

4 Generiske pakker

I forrige kapittel presenterte jeg flere forslag til generiske mekanismer som er lagt fram i den pågående debatten rundt utvidelser av Java. Alle disse forslagene karakteriseres av at de generiske mekanismene ligger på klassenivå, og at de direkte er foreslått som utvidelser til Java.

Jeg skal i dette kapittelet se på en litt annen mekanisme som er annerledes ved at den generiske mekanismen er flyttet til pakkenivå. Utgangspunktet for den er et forslag som ble fremsatt i Simula-sammenheng. Det ble presentert i artikkelen *"A Module-mechanism for Flexible Code Reuse"* skrevet av Krogdahl, Wang, Jensen og Piene [9], men forslaget går egentlig helt tilbake til midten av 80-årene, jamfør *"On Modernizing the Simula Language"* skrevet av Krogdahl [10]. Jeg har her bearbeidet mekanismen, slik at den passer inn i en Java-sammenheng.

Mot slutten av kapittelet vil jeg også, i forbindelse med mekanismen, komme inn på multippel arv, noe som heller ikke er implementert generelt for klasser i nåværende standard Java.

4.1 Kort om mekanismen

En hovedidé med forslaget i [9] er å la pakker (kalt "moduler" i Simula-forslaget) spille en mer sentral rolle enn de gjør i dagens Java, ved at man knytter et begrep om "virtuelle klasser" til disse. Mekanismen har dermed enkelte fellestrekk med Thorup sin i [11], bortsett fra at man i [9] opererer på pakkenivå. Tanken er at man samtidig som man definerer en standard Java-pakke, kan definere en eller flere klasser virtuelle. På bruksstedet av pakka kan man så lage subklasser (av en litt spesiell type) av pakkas virtuelle klasser, og alle ting som er typet med den virtuelle klassen i pakka, skal da forandres slik at de i stedet blir typet med den nye subklassen.

For å illustrere tankegangen, viser jeg her et eksempel på en generisk pakke. I første omgang ser jeg, for enkelhets skyld, bort fra eventuelle problemer med synlighet og lignende. For å skille de generiske pakkene fra standard Java-pakker, har jeg valgt å innføre nøkkelordet `generic`. Dette nøkkelordet oppgis både ved deklarasjon og bruk av pakka. I tillegg viderefører jeg syntaksen som jeg har brukt tidligere hvor jeg setter opp klassene etter hverandre, selv om disse i Java i utgangspunktet skal ligge på forskjellige filer. En generisk pakke kan da se ut som følger:

```
generic package P virtual A, B;
```

```
class A {
    ... A ...      // her menes at A opptrer som type i
    ... B ...      // programteksten.
    ... C ...
}

class B {
    ... A ...
    ... B ...
    ... C ...
}

class C {
    ... A ...
    ... B ...
    ... C ...
}
```

I pakka `P` har jeg definert to virtuelle klasser `A` og `B` som dermed er beregnet på å bli redefinert (altså utvidet og gitt et nytt navn) i et program som bruker den. I tillegg inneholder pakka en "vanlig" klasse `C`. Alle tre klassene har referanser til hverandre, og noen av disse må dermed oppdateres når de virtuelle klassene blir redefinert.

Grunnen til at jeg har valgt en syntaks der man lister opp de virtuelle klassene, er at det blant annet vil gjelde noen spesielle regler for dem; i tillegg til at det kan være klargjørende ved dokumentering og bruk av pakka. Et alternativ kunne være at man skrev "virtuell" der de ble deklarert.

I et brukerprogram er det tenkt at pakka for eksempel skal kunne benyttes som følger:

```
generic import P with K, L;

class K (extends A) {
    ... K ...
    ... L ...
    ... M ...
    ... C ...
}

class L (extends B) {
    ... K ...
    ... L ...
    ... M ...
    ... C ...
}

class M {
    ... K ...
    ... L ...
    ... M ...
    ... C ...
}
```

I dette programmet ønsker man å benytte funksjonalitet fra pakka `P`, og klassene `K` og `L` skal her redefinere `A` og `B`. Klassene `K` og `L` blir dermed *implisitte* subklasser til `A` og `B`, noe som i artiklene er foreslått kalt ”statisk” arv. Av denne grunn trenger man egentlig ikke å skrive dette spesifikt i deklarasjonen av `K` og `L`, men jeg har valgt å legge det inn i parentes for å klargjøre hva som menes.

Programmet inneholder også en klasse `M` som er uavhengig av selve den generiske pakka. Alle klassene på bruksstedet kan referere til `C`, siden den er en ikke-virtuell klasse i pakka.

I programmet blir den generiske pakka `P` ”instansiert” med klassene `K` og `L`, i den betydning at alle forekomster av `A` erstattes av `K` og at `B` tilsvarende erstattes av `L`. I klasse `C` oppdateres også referansene til `A` og `B` slik at de henviser til de nye klassene. Klasse `M` kan være uendret.

Siden mekanismen legger inn noen implisitte arvinger i forbindelse med bruk av pakka (jeg kommer tilbake til dette lenger ned), må det legges enkelte begrensninger på klassene som benyttes. Et par av disse begrensningene kan fjernes hvis man legger inn multippel arv i språket.

- De virtuelle klassene kan ikke ha subklasser i pakka fordi de virtuelle klassene vil erstattes ved pakkas bruk.
- Klassene `K` og `L` får ikke ha noen superklasser siden disse implisitt vil være subklasser av `A` og `B`. Kan fjernes om Java får multippel arv.
- Samme klasse kan ikke forekomme i `with`-lista flere ganger. Kan fjernes om Java får multippel arv.

Før jeg kommer nærmere inn på hvordan mekanismen kan bli implementert i en kompilator, vil jeg først vise et praktisk eksempel på hvordan generiske pakker kan brukes. I kapittel 3.6.1, hvor jeg tok opp problematikken rundt navneendring, presenterte jeg et eksempel rundt oppdatering av objekter. Her vil klassene `Notifier` og `Listener` kunne passe inn i en generisk pakke. I så tilfelle kunne den blitt definert slik:

```
generic package notifyPackage virtual Listener;

class Notifier {
    Listener regObjects[];

    void notify() {...}
}

abstract class Listener {
    // Følgende metode må redefineres i eventuelle subklasser:
    abstract void operation();
}
```

I samme eksempel hadde jeg en brukerklasse `Rectangle` som skulle arve funksjonalitet fra `Listener`. I et brukerprogram vil man hente inn den gene-

riske pakka samtidig som man gir beskjed om at klassen `Rectangle` skal erstatte `Listener`. I dette eksempelet vil jeg ikke fokusere på navneendringen det var snakk om i kapittel 3.6.1, så jeg antar at klassen `Rectangle` har en metode med navn `operation` som redefinerer metoden som er deklarert i superklassen `Listener`. Programmet som henter inn den generiske pakka, blir da som følger:

```
generic import notifyPackage with Rectangle;

class Rectangle (extends Listener) {
    void operation() {...}
}
```

Ved å benytte generiske pakker i dette eksempelet, vil typene som er brukt i klassen `Listener` og `Notifier`, bli oppdatert slik at de passer sammen med klassen `Rectangle`. Med andre ord kan man aksessere attributter i alle tre klassene uten at det er behov for å legge inn castinger.

4.2 Definisjon av mekanismen

Til nå har jeg kort illustrert hvordan man kan se for seg bruk av generiske pakker, og jeg vil nå komme litt mer detaljert inn på hvordan mekanismen *kan* implementeres i en kompilator, og samtidig bruke dette som en definisjon på hvordan mekanismen skal virke.

Videre i teksten vil jeg fortsatt bruke eksempelet med pakka `P` som inneholder klassene `A` og `B`. Når man bruker pakka som skissert over, er altså selve "ideen" å kutte ut hele klasse `A`, og å flette innholdet av `A` inn i `K` med de angitte navneforandringer (og tilsvarende for `B`). Altså for eksempel at `K` skulle bli seende slik ut:

```
class K {
    /* Innholdet av klassen A med A->K og B->L: */
    ... K ...
    ... L ...
    ... C ...
    /* Innholdet av klassen K: */
    ... K ...
    ... L ...
    ... M ...
    ... C ...
}
```

Denne form for sammenfletting kan imidlertid lett føre til Java-kode hvor det blir problematisk med virtuelle bindinger, synlighet, navnekollisjoner osv. Derfor vil det rent teknisk bli greiere hvis kompilatoren internt utfører litt andre transformasjoner; slik at man kan bygge på de mekanismer rundt arv og lignende som allerede er inkludert i språket. Oversettelsen kunne i så fall blitt som følger (der vi antar at navn med `$` er interngenererte navn, som ikke kan brukes i vanlige brukerprogrammer):

Først blir den generiske pakka oversatt (”instansiert”) til en ”vanlig” Java-pakke:

```
package $0$P;

class $1$A {
    ... K ...
    ... L ...
    ... C ...
}

class $2$B {
    ... K ...
    ... L ...
    ... C ...
}

class C {
    ... K ...
    ... L ...
    ... C ...
}
```

Deretter hentes pakka inn som en vanlige Java-pakke, men slik at programmet kobler seg opp til de nye interngenererte navnene.

```
import $0$P;

class K extends $1$A {
    ... K ...
    ... L ...
    ... M ...
    ... C ...
}

class L extends $2$B {
    ... K ...
    ... L ...
    ... M ...
    ... C ...
}

class M {
    ... K ...
    ... L ...
    ... M ...
    ... C ...
}
```

Ved å forme om bruken av den generiske pakka på denne måten, unngår man mange av problemene som oppstår ved direkte sammenslåing av tekstene. Navnekollisjoner, redefinisjoner og lignende blir vel definert gjennom allerede kjente begreper. Denne oversettelsen viderefører også tankegangen som ligger til grunn ved bruk av vanlige pakker i Java.

Et annet resultat av en slik oversettelse er blant annet at deler av ”synlighets-problematikken” vil falle ganske naturlig på plass siden oversettelsen benytter allerede kjente Java-konstruksjoner. En annen ting man oppnår ved å oversette et program på denne måten, er at det naturlig blir definert en ”instans” av den generiske pakka. Jeg kommer mer detaljert inn på dette lenger ned.

Legg merke til at hvert av de interngenererte klassenavnene `$n$<navn>` bare forekommer to ganger, nemlig for eksempel for å gjøre `$1$A` til superklasse av `K`. Dermed er det ikke noen attributter eller uttrykk i programmet som er typet med `$1$A`, og denne klassen kan derfor godt fjernes av en kompilator etter ekspansjonen, samtidig som innholdet, med passelige modifikasjoner, inkorporeres i `K`.

4.3 Mer avansert bruk av generiske pakker

Nå som jeg har presentert hovedideen ved mekanismen, skal jeg se på litt mer avansert bruk av generiske pakker og hvilke følger dette får for utforming av en del detaljer. Blant annet skal ett program kunne ”instansiere” en generisk pakke flere ganger.

Når man i et program importerer vanlige Java-pakker, gir man egentlig kompilatoren bare beskjed om hvilke klasser som skal være tilgjengelig i programmet og hvor man finner dem. Etter en slik import, vil klassene være tilgjengelig på lik linje som andre klasser i programmet.

Siden de generiske pakkene hentes inn samtidig som de blir modifisert, vil hver import føre til en ny, annerledes instans av den generiske pakka. Man vil kanskje i et program ha behov for å hente inn en pakke i flere forskjellige versjoner (for eksempel en liste av personer og en liste av biler), og disse må holdes atskilt siden de representerer forskjellige instansieringer. Merk at man ved to instansieringer av `P` over, også må holde de to `C`-klassene atskilt, da de har fått substituert inn forskjellige navn.

Når en generisk pakke instansieres flere ganger, kan dette altså føre til flertydighet, og det vil derfor være et behov for at man navngir hver instans, slik at man kan skille mellom de forskjellige versjonene. Denne navngivningen har jeg valgt å sette på slutten av importeringssetningen med nøkkelordet `as`, og den er *kun* nødvendig i de tilfeller hvor samme pakke hentes inn flere ganger i samme program.

```
generic import P with K, L as P1;
generic import P with X, Y as P2;
```

Ikke-virtuelle klasser i en generisk pakke, som for eksempel klasse `C` i pakka `P`, vil altså forekomme med samme navn flere ganger. Det vil derfor her være et behov for at man andre steder i programmet, kan spesifisere hvilken pakke klassen `C` ligger i, og dette er også kun nødvendig ved flertydighet. Slike spesifiseringer har jeg valgt å gjøre med dobbelt kolon (inspirert fra C++):

```
P1::C c1 = new P1::C();
```

Denne ekstra syntaksen kan virke noe tungvint, men i praktisk bruk vil det trolig ikke være noe problem da det ikke er så ofte at samme pakke instansieres flere ganger i samme klasse. I tillegg vil jo ikke problemet være aktuelt for de virtuelle klassene. Disse får jo nye navn likevel.

Ved kompilering vil det også noen steder (for eksempel for klassen `C` i `P`), være et behov for å angi hvilken pakke klassen er en del av. De automatisk genererte navnene kan da utvides til å omfatte pakkenavnet. For eksempel kan dette uttrykkes ved:

```
$<pakke>$n$<navn>
```

Denne nye syntaksen vil ikke være synlig for en programmerer da den kun vil benyttes internt av kompilatoren.

4.4 Generiske pakker kontra parametrisert typer

Til nå har jeg presentert de generiske pakkene ut fra hvordan de rent teknisk er definert. Nå vil jeg vise mer hvordan pakkene kan benyttes i praksis, og i denne forbindelse også vise i hvilke sammenhenger vanlige parametriserte klasser (fra kapittel 3) fungerer best og når det kan være behagelig å bruke de generiske pakkene definert over. Jeg tenker meg her at vi bare har enten den ene eller den andre mekanismen tilgjengelig, men jeg skal siden også se på hvordan de to mekanismen kan arbeide sammen.

Jeg vil her presentere to alternative listesystemer, hvorav begge inneholder flere klasser som kan benyttes for å programmere på en liste. I det første alternativet, inneholder pakka en klasse `Link` som representerer selve listeelementet, og denne innholder blant annet en peker til objektet som listeelementet peker på. Klassen `Element` som representerer selve dataobjektet, er definert som virtuell, og den skal med andre ord overstyres ved hver bruk av pakka. Dette systemet kan skisseres med følgende pakke:

```
generic package listSystem1 virtual Element;

class Head {
    Link first;
    // Metoder for å sette inn og ta ut elementer.
    ...
}

class Link {
    Link next;
    Element elem;
    ...
}

class Element {
    // Denne klassen benyttes kun som superklasse for data-
    // elementene i lista, og den inneholder derfor ikke noe
```

```
    // funksjonalitet.
}
```

I programmer som skal benytte dette systemet, må man hente inn pakka samtidig som man definerer hvilken klasse som skal være selve listeelementet. I følgende eksempel ønsker jeg å lage en liste med personer.

```
generic import listSystem1 with Person;

class Person (extends Element) {
    ...
}
```

Som jeg viste tidligere i kapittelet, vil den generiske pakka `listSystem1` bli modifisert noe i det den "instansieres" i brukerprogrammet. Blant annet vil alle referanser til klassen `Element` bli oppdatert til å henvise til klassen `Person`. På denne måten vil man slippe å bruke castinger i koden.

I dette enkle eksempelet er det egentlig ikke så mye å tjene på den nye mekanismen i forhold til parametrisering på klassenivå. Det samme eksempelet kan like lett bli implementert ved hjelp av den parametriserte mekanismen jeg valgte ut i kapittel 3, samtidig som jeg benytter indre klasser.

```
class ListSystem1<Element> {
    class Head {
        Link first;
        ...
    }

    class Link {
        Link next;
        Element elem;
        ...
    }
}
```

Brukerprogrammet kan utnytte listesystemet på følgende måte:

```
class Person {
    ...
}

ListSystem1<Person> list = new ListSystem1<Person>();
...
```

Som man kan se, er de to eksemplene relativt like, men en av fordelene med generiske pakker, er at de fungerer mer i tråd med filosofien rundt pakkebegrepet til Java. I tillegg vil det være behagelig for en programmerer ved importering å kunne parametrisere pakka en gang for alle. Man slipper hele veien å sette inn `Person` som aktuellparameter. Hvis man implementere `listSystem1` ved hjelp av generiske pakker, vil man også ha større mulighet for å kunne lage systemet mer tilpasset et spesielt behov, ved for eksempel å subklasse `Head` og `Link`.

Jeg skal så se på et lignende eksempel der bruk av generiske pakker har flere fordeler. Hvis man i den generiske pakka ikke har en egen klasse `Element` som skal representere objektene som skal kunne lagres, men heller velger å la klassen for de aktuelle objektene være en virtuell subklasse av `Link`, blir situasjonen noe annerledes. Listesystemet vil da bli enklere:

```
generic package listSystem2 virtual Link;

class Head {
    Link first;
    // Metoder for å sette inn og ta ut elementer.
    ...
}

class Link {
    Link next;
    ...
}
```

Ved bruk av dette systemet, vil en aktuell klasse `Person` arve funksjonaliteten fra `Link`, samtidig som man får uttrykt mer naturlig at det faktisk er objekter av typen `Person` som skal være elementer i lista. Et eksempel på bruk av `listSystem2` kan være:

```
generic import listSystem2 with Person;

class Person (extends Link) {
    String name;
}

...

Head h = new Head();
...
// Det vil nå være mulig å aksessere variablen "name" direkte
// selv om "first" i utgangspunktet er typet med "Link".
h.first.name = "Ola Hansen";
```

Et slikt listesystem vil umiddelbart ikke være like enkelt å implementere ved hjelp av parametriserte typer. Det er mulig, men det blir noe krunglete. Jeg venter imidlertid med å vise hvordan en slik implementasjon kan gjøres med parametriserte klasser, siden eksempelet i neste avsnitt illustrerer problematikken vel så godt.

Generiske pakker vil i flere tilfeller naturlig kunne bygge på hverandre, og for å vise noe av fleksibiliteten som mekanismen har, kan jeg vise en interessant bruk der jeg benytter `listSystem2` til å programmere `listSystem1`. Dette kan gjøres slik:

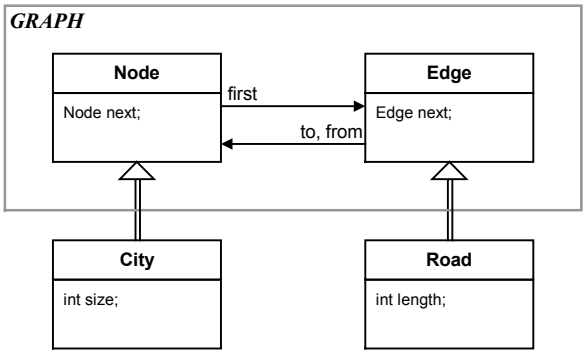
```
generic package listSystem1 virtual Element;

generic import listSystem2 with Link1;
```

```
class Link1 (extends Link) {
    Element elem;
}

class Element {
}
```

4.5 Simultan arv, Graf-eksempel

I artikkelen av Krogdahl og kollegene [9] presenterer de et eksempel som viser styrken til deres nye mekanisme. Dette er et eksempel rundt programmering av grafer ved hjelp av en pakke `graph` som inneholder klassene `Node` og `Edge`. Denne pakka vil jeg så benytte i forbindelse med programmering av en bystruktur, hvor `Node` "utvides" til `City` og `Edge` til `Road`.

Ett av de essensielle poengene med dette eksempelet, er at både `Node` og `Edge` inneholder en god del funksjonalitet som skal benyttes i subklassene. Derfor egner det seg dårlig å sende dem inn som vanlige klasseparametere. I tillegg har begge klassene gjensidig referanse til hverandre, og disse referansene skal også beholdes, men types om i subklassene som defineres på bruksstedet.



Figur 4.1 Klassestruktur for bysimulering, `City` og `Road`.

Klassestrukturen for eksempelet er gitt i Figur 4.1, hvor jeg har valgt å uttrykke den "statiske arven" med en dobbelt pil. Pakka for grafen kan kodemessig skisseres som følger hvor klassene `Node` og `Edge` altså er definert som virtuelle.

```
generic package graph virtual Node, Edge;

class Node {
    Node next;  // F.eks. neste i en liste av alle noder.
    Edge first; // Første kant for denne noden
    // inneholder en del funksjonalitet for noden
    ...
}

class Edge {
```

```
    Edge next; // Neste kant fra denne node.
    Node from, to;
    // inneholder en del funksjonalitet for kanten.
    ...
}
```

I brukerprogrammet hentes pakka inn, samtidig som det gis beskjed om at klassen `City` skal erstatte `Node` og klassen `Road` skal erstatte `Edge`.

```
generic import graph with City, Road;

class City (extends Node) {
    int size;
    // Kan her greit skrive "first.length".
    ...
}

class Road (extends Edge) {
    int length;
    // Kan her greit skrive "to.size".
    ...
}
```

Dette eksempelet kunne for så vidt også blitt implementert med parametriserte klasser og vanlig arv, men da med mer komplisert og mindre intuitiv kode. Man må da ha to generiske parametere på begge klassene.

Ved hjelp av parametriserte typer kan klassene deklareres som følger:

```
class Node<SubNode extends Node, SubEdge extends Edge> {
    SubNode next;
    SubEdge first;
    ...
}

class Edge<SubNode extends Node, SubEdge extends Edge> {
    SubEdge next;
    SubNode to, from;
    ...
}
```

I et program som skal benytte disse klassene, blir deklarasjonen av `City` og `Road` noe enklere, men man må oppgi parameterne to ganger, og programmereren må selv passe på at man gir samme parametere begge steder:

```
class City extends Node<City, Road> {...}
class Road extends Edge<City, Road> {...}
```

Som man kan se i koden over, er det en del enklere å uttrykke det samme hvis man kan benytte generiske pakker. Blant annet slipper man å innføre nye ”klassenavn” som `SubNode` og `SubEdge`, samtidig som man får gitt parameterne på ett sentralt sted. I tillegg faller de generiske pakkene som sagt bedre inn i Java sin filosofi rundt pakkebruk.

4.6 Generiske parametere på generiske pakker

Som jeg nå har vist, kan det være et behov både for parametriserte klasser og generiske pakker. Mekanismene har styrker på hver sine områder, men disse mekanismene kan også kombineres. På slutten av artikkel [9] nevner forfatterne at det kanskje kan være fordelaktig å utvide de generiske pakkene med parametere slik de brukes på parametriserte klasser. Dette er fullt mulig, og man kan da tenke seg at disse parameterne også kan beskrankes med for eksempel nøkkelordene *extends* og *implements* og/eller *where-clauses*. En generiske pakke kan da for eksempel deklareres slik:

```
generic package P virtual A, B <C extends D> {
    ...
}
```

Ved å utvide de generiske pakkene på denne måten, vil det blant annet være enklere å benytte dem med ferdig definerte klasser. Over skisserte jeg to eksempler med bruk av en generisk pakke `listSystem`. I den første versjonen måtte en aktuell "elementklasse" være definert som en subklasse av `Element`. Hvis man benytter parametere som over, vil ikke dette lenger være nødvendig. Med andre ord kan bruk av pakker bli noe mer fleksibel.

Man vil alltid i et programmeringsspråk måtte avveie ønsker om ny funksjonalitet opp mot at dette kan gjøre et språk vanskeligere og kompilatorene større og dermed tregere. Innføring av vanlige parametere på generiske pakker, er nok en utvidelse som må vurderes om er hensiktsmessig i forhold til hvor nyttig den er.

Med tanke på omfanget av denne oppgaven og at utvidelsene av Java ikke bør være alt for omfattende, har jeg valgt å ikke gå nærmere inn på denne tilleggsfunksjonaliteten. Jeg har raskt skissert hvordan parametere på de generiske pakkene kan fungere, men jeg vil videre i denne oppgaven gå ut i fra at jeg ikke har dem tilgjengelig. Med andre ord tillater jeg generiske parametere på klasser (som presentert i kapittel 3), men ikke på de generiske pakkene.

Siden jeg i denne oppgaven har valgt å innføre to forskjellige generiske mekanismer, vil det være veldig viktig å sjekke ut at disse kan "leve" parallelt uten at de ødelegger for hverandre. En ting man da må avklare, er hvordan parametriserte klasser og generiske pakker skal arbeide sammen, og det viktigste da er blant annet å avgjøre om virtuelle klasser får lov å være parametriserte, og hvordan dette i så fall skal fungere.

Ser man på noen eksempler, viser det seg at man raskt får behov for klasser i generiske pakker som både er virtuelle og som har generiske parametere. Vi kan altså for eksempel få følgende situasjon:

```
generic package P virtual A;

class A<K> {
    K k = new K();
}
```

```
class B<L> {
}
```

Som man kan se i eksempelet, har jeg valgt at man ikke oppgir de generiske parameterne ved definering av pakka, men først når man deklarerer selve klassen. Dette valget har jeg gjort da det trolig ikke vil være noe grunn til å måtte oppgi de *samme* parameterne begge steder. Ved bruk av denne pakka, synes det da fornuftig å angi de aktuelle parameterne til `A` og `B` slik:

```
generic import P with AA;

class AA (extends A<KK>) {
}

...

B<LL> Bll; // Virker helt som vanlig.
```

Ved definisjonen av `AA`, gjelder her de samme regler som for de generiske typene fra kapittel 3. Den generiske parameteren `KK` blir her dermed en aktuellparameter til den generiske klassen `A`. Man kan også la `AA` bli en generisk klasse videre, og det vil derfor være fullt lovlig å deklarere `AA` slik:

```
class AA<KK> (extends A<KK>) {
}
```

Ved å la de to generiske mekanismene samarbeide på denne måten, blant annet ved å integrere så mye som mulig av den generiske syntaksen, vil det trolig være ganske greit for en bruker å bli fortrolig med disse nye utvidelsene av språket.

4.7 Multippel arv

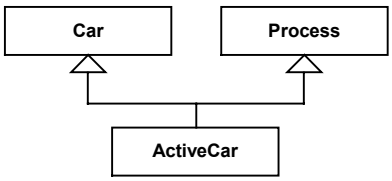
En ide rundt ”ideell” programmering, som også Krogdahl [9] vektlegger, er tanken om at man bør ha så gode og generisk fleksible biblioteker at man i et program rett og slett skal kunne ”plukke” funksjonalitet og flette disse sammen i brukerklasser. Brukerprogrammene og -klassene kan i mange tilfeller dermed bli ganske korte; siden *mye* av funksjonaliteten blir hentet fra biblioteker.

Man kan for eksempel tenke seg at man ønsker å lage en trafikksimulering, hvor man i et program henter inn en klasse `Process` fra en pakke `Simulation` og en klasse `Car` fra en pakke `Traffic`. Videre kan man tenke seg at man ønsker å opprette en klasse `ActiveCar` som arver både fra klassene `Process` og `Car`. I et Java utvidet med multippel arv, kunne dette bli uttrykt med følgende kode.

```
class ActiveCar extends Process, Car {
    ...
}
```

Et objekt av klassen `ActiveCar` er nå en subklasse av både `Process` og `Car`, og objektet kan dermed fungere på alle plasser hvor det forventes et objekt av en av de to superklassene.

For å kunne få til en slik form for programmering, må programmeringsspråket støtte en eller annen form for multippel arv. I utgangspunktet gjør ikke Java dette for klasser, men jeg vil her presentere kort hvordan dette naturlig kunne inkorporeres i språket.



Figur 4.2 Klassestruktur med `ActiveCar`.

Ved innføring av multippel arv vil man også kunne fjerne enkelte av begrensningene jeg presenterte i forbindelse med de virtuelle klassene i de generiske pakkene.

Siden multippel arv er en omstridt mekanisme, vil jeg først presentere forskjellige sider ved mekanismen, og som utgangspunkt har jeg tatt for meg implementasjonen som brukes i språket C++. Jeg vil også se kort på en alternativ implementasjon som tar utgangspunkt i noe som kan kalles ”statisk multippel arv”, en mekanisme som forfatterne [9] mener bør vurderes som et alternativ til full multippel arv da den har en del av fordelene ved multippel arv uten å få med alle ulempene. Jeg kommer noe tilbake til dette i kapittel 4.8.

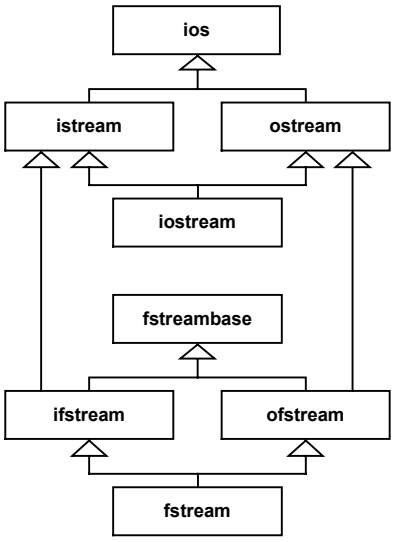
Denne hovedoppgaven skal i hovedsak vurdere generiske typer, og fokus her er derfor rettet mot dette temaet og ikke mot multippel arv som sådan. Denne omtalen blir derfor ganske kort, og den er forsøkt rettet mot de sider av multippel arv som kan få betydning ved utforming og bruk av generiske mekanismer.

4.7.1 Generelt om multippel arv

I sin bok *”The design and evolution of C++”* [12], har Bjarne Stroustrup skrevet litt rundt innføringen av multippel arv i C++. Blant annet har han skrevet litt om erfaringene han har fått og litt rundt motstanden han har møtt i prosessen.

I tiden før Stroustrup la inn multippel arv i C++, var han bekymret for at bibliotekklassene var i ferd med å vokse seg for store og uoversiktlige. Ved å innføre multippel arv, fikk han strukturert klassene bedre og dermed forenklet mange av dem. Som et eksempel, trekker han fram en del av klassene som brukes for å lese og skrive til systemet. Figur 4.3 viser hvordan strukturen ble

etter at multippel arv var innført. Uten multippel arv var klassestrukturen vesentlig mer komplisert.



Figur 4.3 Klassestruktur for IO i C++

Grunntanken er at man ønsker at en ny klasse som skal kunne arve egenskapene til flere andre klasser, på en slik måte at objekter som blir generert fra resultatklassen, kan fungere som objekter av begge superklassene. Et enkelt eksempel er klassen `ActiveCar` som jeg presenterte over. Denne arver både fra klassen `Process` og `Car`.

Når man tillater at en klasse skal arve attributtnavn og kode fra flere klasser, kan det raskt oppstå navnekollisjoner og andre flertydigheter. Dette krever at det defineres klare regler som løser dette. Senere i kapittelet vil jeg komme tilbake til noen av problemene og også skissere hvordan de kan løses.

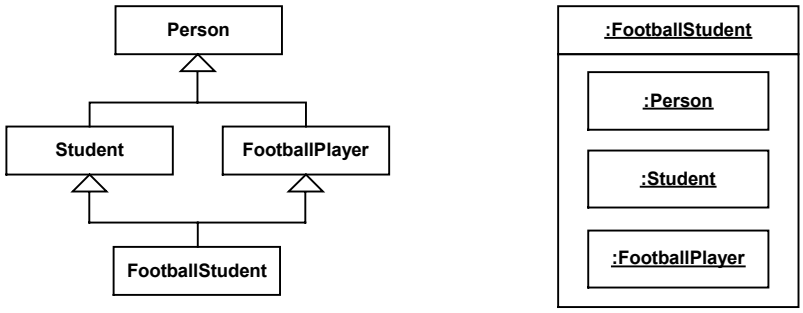
4.7.2 Virtuelle og uavhengige baseklasser

Når man arver fra flere klasser, kan man komme opp i situasjoner hvor klassestrukturen blir formet som en ”ruter”. Et karakteristisk eksempel kan være et program som jobber med forskjellige typer personer hvor alle klasser arver fra en felles superklasse `Person`.

Et konkret eksempel kan være en klasse `FootballStudent` som både vil arve fra `Student` og `FootballPlayer`. Begge disse klassene er igjen subklasser til den felles superklassen `Person`. Denne form for arv kalles ofte virtuell arv siden et objekt av `FootballStudent` kun skal inneholde én `Person`-del.

Figur 4.4 viser hvordan klassestrukturen er og hvordan et objekt av klassen `FootballStudent` blir bygget opp i minnet på maskinen. UML inneholder ingen konkret notasjon for å vise hvordan objekter er bygget opp ”innven-

dig". Derfor har jeg tatt utgangspunkt i UML, men komponert figuren av objektet delvis selv.



Figur 4.4 Klassestruktur for *virtuell arv* og skisse av hvordan et objekt er representert i minnet på maskinen.

Hvis man skal opprette en slik struktur, må man kunne spesifisere at klassen `Person` skal arves bare én gang ned i eventuelle felles subklasser (som `FootballStudent`), og i C++ gjøres dette med nøkkelordet `virtual`. Deretter arver klassen `FootballStudent` fra begge disse to klassene på vanlig måte. I Java kunne man uttrykke dette med følgende syntaks som er inspirert av C++.

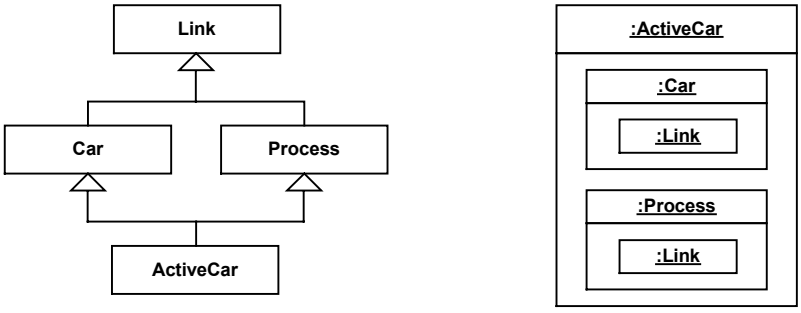
```
class Student virtual extends Person {...}

class FootballPlayer virtual extends Person {...}

class FootballStudent extends Student, FootballPlayer {...}
```

Ved å spesifisere at klassen `Person` skal arves virtuelt, oppnår man at det kun blir opprettet én `Person`-del for hvert objekt av klassen `FootballStudent`.

I eksempelet over var det ønskelig med kun én utgave av den felles superklassen. I andre tilfeller er det ønskelig at det i hvert objekt opprettes to uavhengige utgaver av en eventuell felles superklasse.

Som et eksempel på det, har jeg igjen tatt utgangspunkt i simulering av trafikk. Her kan man tenke seg at man har en klasse `Car` som skal kunne stå i køer. Køen registreres som en liste, og klassen registreres derfor som en subklasse av `Link`. I tillegg ønsker vi oss simuleringsprosesser, og disse skal også kunne registreres i en liste av planlagte hendelser. Klassen `Process` registreres derfor også som en subklasse av `Link`, men disse listene skal være helt uavhengige av hverandre.

Den multippelt arvede klassen `ActiveCar` skal dermed kunne stå både i en kø av biler og i en kø over planlagte hendelser. Strukturene blir da som på Figur 4.5. Når man oppretter klassestrukturen uten å spesifisere at klassen `Link` arves virtuelt, vil `Car`-delen og `Process`-delen få hver sin uavhengige `Link`-del.



Figur 4.5 Klassestruktur for *uavhengig arv* og skisse av hvordan et objekt er representert i minnet på maskinen.

I en implementasjon av multippel arv, vil det være viktig at man klarer å uttrykke begge disse former for arving. Dette er viktig fordi forskjellige problemtyper trenger forskjellige klassestrukturer for å løses mest hensiktsmessig.

4.7.3 Multippel arv ved hjelp av delegering

Det originale designet for multippel arv i C++, som ble presentert på *"European UNIX Users' Group (EUUG)"*-konferanse i Helsinki i mai 1987 [13], inneholdt en del om delegering. Noen ser faktisk på multippel arv som en for problemfylt og uklar mekanisme (jeg kommer tilbake til noen av disse momentene lenger ned), og viser derfor til delegering som et alternativ.

Delegering er en teknikk hvor man i en klasse som skal arve fra flere andre klasser, legger inn de aktuelle superklassene som "delobjekter" og hvor man samtidig inkluderer metoder for å videresende alle metodekall inn til det riktige delobjektet. Det er vanskeligere å benytte teknikken for å automatisk videreføre verdier til variable, men dette kan løses ved å inkludere metoder for å sette og returnere verdien for hver variabel.

Nedenfor har jeg igjen brukt eksempelet rundt `ActiveCar`, men hvor jeg nå erstatter multippel arv med delegering. Jeg har minimalisert eksempelet slik at jeg blant annet har fjernet all kode som beskriver hvordan man oppretter delobjektene. Det viktige nedenfor er å få fram det grunnleggende i hvordan metodekall blir videresendt. Nedenfor har jeg satt opp en skisse til de to klassene `Process` og `Car` som man kan tenke seg å arve fra.

```
class Process {
    void startSimulation() {...}
    ...
}
```

```
 class Car {
    void setDriver(String name) {...}
    ...
}
```

I klassen `ActiveCar` som skal arve fra de to andre, vil det være en peker til et objekt av hver av de andre klassene og samtidig metoder for å videresende alle nødvendige kall.

```
class ActiveCar {
    Process process;
    Car car;

    void startSimulation() {
        process.startSimulation();
    }

    void setDriver(String name) {
        car.setDriver(name);
    }

    ...
}
```

Lenger opp nevnte jeg at man kunne arve med både virtuelt og uavhengige baseklasser. Ved virtuell arv ble den felles superklassen kun representert én gang for hvert hovedobjekt. Motsetningen var å jobbe med uavhengige baseklasser. Begge disse måtene kan man få til ved hjelp av delegering, men det blir raskt veldig tungvint og ganske innviklet.

Java støtter i nåværende versjon kun enkel arv på klasser, men multippel arv på interfacer. Ved å kombinere dette på en fornuftig måte, kan man få til en relativt god delegeringsmekanisme der man langt på vei kan tenke som om man har fullverdig multippel arv. Men jeg som mange andre, mener at den fortsatt blir for komplisert og ”ad hoc”. Jeg vil derfor i denne oppgaven ta utgangspunkt i at man innfører en fullverdig multippel arv til språket.

4.7.4 Problemer rundt multippel arv

Det vanligste problemet man kan støte borti når man arver fra flere klasser, er flertydighet. Hva skal skje hvis begge klassene man arver fra, inneholder en metode med samme navn og parametere, men med forskjellig innhold? Uten spesifikt å angi hvilken metode som skal velges, vil ikke kompilatoren vite hva den skal gjøre og den vil protestere. En løsning er å la kompilatoren nekte å kompilere flertydige programmer, men dette er en løsning som raskt vil kunne komme til kort.

I det flertydige eksempelet under, hvor `C` arver både fra `A` og `B`, vil ikke kompilatoren være i stand til å velge hvilken forekomst av metoden `f` som skal kjøres.

```
class A {
    void f();
}

class B {
    void f();
}

class C extends A, B {
    // Klassen C inneholder ikke f().
}

...

void g() {
    C c;
    c.f();  // Kompilatoren kan ikke bestemme hvilken f()
            // som skal kalles.
}
```

Stroustrup foreslo i sitt første forslag til multippel arv i C++ at kompilatoren automatisk skulle velge den `f` som lå i klassen `A` siden denne tekstlig kom først i lista over klasser det arves fra, men denne løsningen hadde han dårlig erfaring med, og valgte derfor at programmereren selv måtte spesifisere, i de tilfeller hvor det var flertydighet, hvilken klasse metoden skulle hentes fra. Dette gjør man ved å spesifisere klassen ved selve kallet `A::f`, hvor `A` er klassen som inneholder `f`. Denne løsningen hadde han bedre erfaring med.

I C++ har man forøvrig både virtuelle og ikke-virtuelle funksjoner, mens Java (nesten) bare har virtuelle. Derfor er egentlig problemstillingen ikke fullt så aktuell for Java, men jeg ville likevel nevne det. Uansett har jeg valgt at flertydighet løses ved at man må spesifisere hvilken klasse man ønsker å hente attributtet fra.

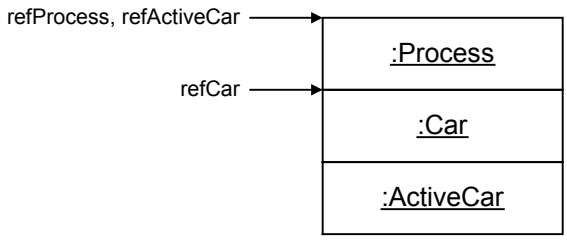
I tillegg til problemer rundt mulige navnekollisjoner, er det et klassisk problem som nesten alltid blir tatt frem når man diskuterer multippel arv, og det omhandler beregning av relativadresser ved kompilering av klasser.

La oss tenke oss at klassen `Process` kompileres for seg selv. Adressene for å aksessere variable og metoder i objekter av klassen `Process`, blir oppgitt relativt ut i fra starten av objektet. Det samme vil det være for klassen `Car`.

Hvis vi nå oppretter klassen `ActiveCar` som arver både fra `Process` og `Car`, vil det være et problem med adressering av variable i objekter som er generert ut i fra denne klassen. Se Figur 4.6 for illustrasjon hvordan et objekt av klassen `ActiveCar` vil være representert i maskinen. Standard UML kan dessverre ikke uttrykke hvordan et objekt er bygget opp. Derfor har jeg illustrert dette selv, men hvor jeg benytter UML i bunn.

Som jeg nevnte over, pleier man å beregne relative adresser som hjelp for å kunne aksessere metoder og variable i objekter. Disse relative adressene i tillegg til den eksakte minneadressen til starten av objektet, danner til sammen den eksakte adresse til hvert enkelt attributt.

Tenk på et objekt som er generert ut i fra klassen på Figur 4.6. Her er `Process`-delen av objektet plassert først, og det vil med andre ord ikke være noen problemer med å få tak i attributtene fra `Process`-delen ut i fra en referanse som peker på starten av objektet. Det vil heller ikke være noe problem å beregne de riktige adressene til attributter i klassen `ActiveCar` fordi begge klassene `Process` og `Car` er kjent ved kompilering av denne. Både pekere typet med `Process` og med `ActiveCar` kan derfor peke til starten av objektet.



Figur 4.6 Et `ActiveCar`-objekt generert med multippel arv.

Det vanskelige oppstår når man skal beregne de relative adressene til attributtene i `Car`-delen av et objekt av klassen `ActiveCar`. Siden klassen `Car` kan være kompilert på forhånd, og siden det kan eksistere objekter av bare klassen `Car`, vil også denne klassen aksessere sine attributter ved hjelp av relativadresser fra starten av sin del av objektet. Hvis pekeren til et `ActiveCar`-objekt nå peker på starten, vil det bli feil når `Car` skal hente sine verdier. De absolutte minneadressene som blir beregnet, vil være feil i forhold til attributtene i `Car`.

Dette problemet kan for eksempel løses på samme måte som blant annet Stroustrup har gjort i C++ [12], nemlig ved å la `refCar`-pekere peke på starten av `Car`-delen isteden for å peke på starten av selve hovedobjektet. Ved å gjøre dette og ved å la klassen `ActiveCar` legge til eller trekke fra en konstant størrelse under tilordning, størrelsen på `Process`, kan man nå attributter i hele objektet selv om man tillater separatkompilering av klasser.

4.8 Statisk multippel arv

Jeg introduserte multippel arv med å si at dette er en mekanisme som er mye diskutert, både fordi det er delte meninger om hvor hensiktsmessig mekanismen er, i tillegg til at den byr på vanskeligheter som navnekollisjoner, beregning av relative adresser osv.

Jeg har ved hjelp av generiske pakker, vist at det kan være et behov for multippel arv hvis man velger den definisjonen som jeg har presentert tidligere i dette kapittelet. I kapittel 5 vil jeg presentere et litt større praktisk eksempel, og her vil også multippel arv kunne gjøre programmeringen mer behagelig.

Jeg vil derfor si at det er et behov for å kunne arve fra flere klasser samtidig, men som det blir antydet i [9], kan det kanskje være tilstrekkelig med en noe begrenset mekanisme. Som et alternativ til mekanismen slik den er i C++, kan man se for seg noe som kan kalles ”statisk multippel” arv. Dette er en videreføring av tankegangen rundt ”statisk arv” som jeg har introdusert med generiske pakker.

Hovedideen rundt ”statisk arv”, er altså en mekanisme hvor man tenker seg at en brukerklasse tekstlig blir flettet sammen med superklassen sin, samtidig som typene i superklassen og hele dens omgivelse blir oppdatert slik at de stemmer overens med den nye klassen. Denne ideen kan man bygge videre og tenke seg at en klasse kan flettes sammen med flere superklasser.

Ved å gjøre det på denne måten, kan man arve funksjonalitet fra flere klasser, samtidig som man unngår mye av den potensielle flertydigheten når det gjelder typer på pekere. Man vil da ha fordelen av å kunne hente kodebiter til en klasse fra forskjellige steder, samtidig som man under utførelsen ikke vil sitte med et hierarki av klasser med multippel arv, siden man ved statisk arv slår sammen de involverte klassene til én allerede under kompilering.

Det vil fortsatt kunne forekomme navnekollisjoner, men dette kan løses ved å definere klare konvensjoner for hvordan flertydigheten skal behandles. Det enkleste er rett og slett å lage en syntaks for å kunne angi hvilke av flere alternativer man mener.

På den ene siden vil altså en mekanisme som statisk multippel arv, være en enklere og mindre problemfylt mekanisme, men på den annen side vil den ikke kunne bli like kraftig som full multippel arv. Blant annet vil det ikke være mulig å få til virtuell multippel arv hvor det vil være én felles del av en felles superklasse. Siden en klasse blir tekstlig sammensatt av koden i superklassen, blir de felles baseklassene automatisk uavhengige av hverandre.

Java, i motsetning til mange andre språk, støtter bruk av interfacer. Siden dette kun er abstrakte klasser *uten* noen implementasjon, er det en god del enklere å tillate multippel arv av disse, og dette gjøres da også i standard Java. Dette bør derfor videreføres for de generiske pakkene, uavhengig om man tillater multippel statisk arv eller ikke for virtuelle klasser.

Det bør naturlig være mulighet til å kunne definere virtuelle interfacer på samme måte som for klasser, og i et brukerprogram bør man kunne hente inn flere generiske pakker og multippelt arvede interfacer statisk fra alle disse. Man kan for eksempel tenke seg at det finnes to generiske pakker `P1` og `P2`, hvor `P1` inneholder et virtuelt interface `I` og `P2` innholder et virtuelt interface `J`. I et brukerprogram bør det da være lov å benytte disse på følgende måte:

```
generic import P1 with C;
generic import P2 with C;

class C (implements I, J) {
    ...
}
```

I dette tilfellet vil det som opprinnelig var definert som et interface, bli erstattet med en klasse, men dette bør naturlig gå uten videre problemer. Det vil i brukerprogrammet også være helt greit å lage "subinterfacer" av både `I` og `J`.

Merk her at det på samme måte som for virtuelle klasser, ikke vil være lov å lage "subinterfacer" eller klasser av de virtuelle interfacene inne i den generiske pakka. Disse vil jo bli erstattet ved pakkas bruk, og slike avhengigheter vil dermed bli ugyldige etter at en generiske pakke er instansiert.

Jeg vil mot slutten av oppgaven knytte noen kommentarer til om man i eksempelet i kapittel 5 kunne klare seg med statisk multippel arv, men i kapittel 5 for øvrig vil jeg anta at jeg har full multippel arv tilgjengelig.

4.9 Sammendrag

Jeg har i dette kapittelet skissert en generisk mekanisme kalt "generiske pakker" hvor man på pakkenivå innfører en form for virtuelle klasser. Dette kan være gunstig i mange tilfeller hvor klasser typisk hører sammen som en enhet, altså en pakke. Denne mekanismen er ikke ment å være et alternativ til parametriserte klasser, men skal være en sidestilt mekanisme som kan være mer hendig å bruke i en del tilfeller.

I forrige kapittel hvor jeg beskrev vanlig parametriserte klasser, var disse stort sett implementert og testet i bruk. Mekanismen i dette kapittelet er ikke implementert i Java, men jeg har tatt utgangspunkt i relativt kjente og enkle begreper og benytter for eksempel arv som allerede eksisterer i språket. Blant annet av den grunn, regner jeg med at mekanismen som sådan relativt greit skal kunne implementeres, men dog kanskje med noen forandringer i forhold til hva jeg har beskrevet. Lignende mekanismer er imidlertid testet implementert i Simula [27].

I tillegg har jeg, ut i fra behov som har dukket opp, valgt å ta utgangspunkt i at Java også utvides med multippel arv etter samme modell som Stroustrup har brukt i C++ [12] og [13]. Denne utvidelsen vil jeg som sagt også ha behov for i neste kapittel hvor jeg skal gå gjennom konkrete programeksempler og teste ut de nye mekanismene, men jeg vil til slutt også vurdere i hvilken grad man kunne klare seg med en enklere form, nemlig statisk multippel arv.

5 Design-patterns

Denne oppgaven har så langt beskrevet en del framsatte forslag til generiske mekanismer i Java, og jeg har vurdert dem blant annet ved å se på hvor godt de løser visse problematikker rundt typer og hvor gode muligheter man har for å kunne spesifisere de generiske typene. Jeg har også diskutert om de foreslåtte mekanismene lar seg kombinere slik at de kan benyttes parallelt.

I dette kapittelet skal jeg gå nærmere inn på sammensatt bruk av de to generiske mekanismene fra kapittel 3 og 4, og i dette arbeidet har jeg valgt å jobbe med såkalte design-patterns og se på i hvilken grad de generiske mekanismene blant annet klarer å fange inn de essensielle problemene rundt typer for hvert pattern.

Først vil jeg presentere kort hva design-patterns er, før jeg presenterer tre av dem mer i detalj. For hver av de tre utvalgte patternene vil jeg gi et eksempel på typisk bruk, og mot slutten av kapittelet vil jeg konstruere et større eksempel hvor jeg altså kombinerer alle tre. Med dette som bakgrunn, vil jeg diskutere forskjellige sider av de generiske mekanismene, hvor greit det er å kombinere dem og også komme litt tilbake til vurderingen av multippel arv.

5.1 Hva er design-patterns?

Christopher Alexander og hans kolleger [24] var i 1977 kanskje de første som kom med ideen om å snakke om et pattern-språk, hvor man generaliserer handlinger og strukturer i såkalte mønstre eller pattern. Blant annet har Christopher prøvd å beskrive hva et pattern egentlig er, og følgende sitat viser hvor abstrakt det kan være:

> *”Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.”*

Alexander snakker her neppe om gjenbrukbar programkode, men han snakker om generelle mønstre innenfor arkitektur til bygninger og byer. Likevel har tankegangen rundt pattern blitt tatt i bruk i ulike profesjoner, og deriblant også innenfor design av programvare.

I boka *”Design patterns”* [2] forsøkte forfatterne å få tak i essensen av problemer som dukker opp igjen og igjen når man programmerer. Disse trenger ofte grundig gjennomtenkning før man kan sette seg ned og programmere. Forfatterne har funnet fram til 23 slike problemer og gjort deler av dette tankearbeidet for oss. De har tatt dem for seg én etter én og forsøkt å skrive ned

hva problemene består i, hvordan man kan løse dem mest hensiktsmessig under forskjellige forhold, hva man ikke bør gjøre, og så videre.

Design-patterns er per definisjon ganske generaliserte og abstrakte. Hvordan man implementerer en stakk i et objektorientert språk, vil typiske ikke bli regnet som et pattern. Det er for konkret og rett fram. Likevel er det få klare regler for hva som må til for at en problemstilling med en eller flere typiske løsninger skal kunne kalles et design-pattern. Hele tankegangen rundt design-pattern i [2] er imidlertid knyttet til objektorientert tankegang, og både problemene og løsningene blir beskrevet i objektorienterte begreper.

Noe av årsaken til at de forskjellige patternene i boka kun er presentert som "tankeverk", er at de er så abstrakte slik at den konkrete implementasjon vil variere mye fra situasjon til situasjon. De blir derfor i [2] ikke presentert med komplett kode, men kun med antydninger om hvordan de kan implementeres i forskjellige sammenhenger. Man beskriver hvordan de fungerer, hva de kan brukes til, eksempler på bruk osv. Design-patterns er delvis uavhengig av språk, men er altså i [2] sterkt knyttet til objektorienteringen som sådan.

Alle patternene som er definert i boka, presenteres ved hjelp av fire essensielle elementer:

1. **Navnet** består av ett til to ord som kort skal beskrive designproblemet og dets løsning. Navnet gjør det enklere å referere til den bestemte designkonstruksjonen, og vi får mulighet til å utføre design på et høyere abstraksjonsnivå.
2. **Problemet** beskriver de situasjoner der det kan være aktuelt å benytte patternet. Problemet skisseres ofte ved hjelp av klasse- og objektstrukturer, men det kan for eksempel også være tips om hvordan algoritmer kan representeres via objekter.
3. **Løsningen** beskriver de elementene som designen er bygget opp av, deres struktur og ansvarsområder. Løsningen er som regel representert med en mal som skal kunne benyttes i forskjellige situasjoner.
4. **Konsekvensene** er positive og negative følger av å ta i bruk patternet på forskjellige måter. Konsekvensene er en veldig viktig del ved vurdering av hvordan et problem bør angripes.

I innledningen av [2] sies det at det generelt er umulig å implementere design-pattern som ferdige bibliotecksklasser. Dette gjelder nærmest som en definisjon i den forstand at problemer der løsningen greit lar seg programmere som en separat bit som kurant kan taes inn og løse et problem der det er behov for det, neppe bil bli betegnet som et design-pattern.

Hvor umulig det er å programmere et design-pattern som en ferdig programbit, vil imidlertid variere mye, både fra pattern til pattern, og ikke minst med hvilke mekanismer man har i det aktuelle programmeringsspråket. Det har blant annet i språket Beta, blitt implementert flere design-pattern som senere er blitt lagt inn i biblioteker [28].

Et viktig poeng i dette kapittelet er derfor å se om de mekanismene jeg har sett på til nå, kan gjøre det enklere å implementere slike pattern på en tilfredsstillende måte i Java.

En av vanskelighetene er at løsningen et pattern foreskriver, ofte vil bli flettet inn i et brukerprogram på forskjellige steder, og kanskje på forskjellige måter fra bruk til bruk. For at en generisk mekanisme skal lykkes her, må den derfor være mest mulig fleksibel med tanke på hvordan de generiske klassene/pakkene kan hentes inn og kombineres.

Jeg vil videre i dette kapittelet presentere tre forskjellige design-patterns hvor jeg for hvert pattern viser hvordan det kan brukes i en konkret kontekst. Etterpå setter jeg sammen de forskjellige eksemplene i forbindelse med at jeg vil lage et system for å vise og oppdatere klokker på skjermen. Til slutt prøver jeg å trekke noen konkusjoner ut i fra erfaringene jeg har fått.

Ved presentasjonen av patternene, tar jeg utgangspunkt i klassene og strukturene som blir brukt i boka *"Design Patterns"* [2]. Flere av disse strukturene kan variere noe når man leser forskjellige bøker, men jeg har som sagt valgt å ta utgangspunkt i beskrivelsen i [2].

I eksempelkoden vil jeg der det passer anta at standardbibliotekene jeg bruker fra Java, er implementert ved hjelp av enten parametriserte klasser eller generiske pakker. Jeg vil dermed for eksempel ha mulighet til å instansiere en vektor med `new Vector<Person>()`, og slipper dermed unna med castingene som jeg ellers hadde trengt.

5.2 Observer

Det første patternet jeg vil se på, *Observer*, fungerer som et overliggende "lag" som kan brukes for å knytte et antall objekter (kalt "observere") sammen med et annet objekt (kalt "subject") som har data som observerne er interessert i. Man har altså objekter som er avhengig av data i et felles objekt, og disse ønsker å bli fortalt når "dataobjektet" endrer seg.

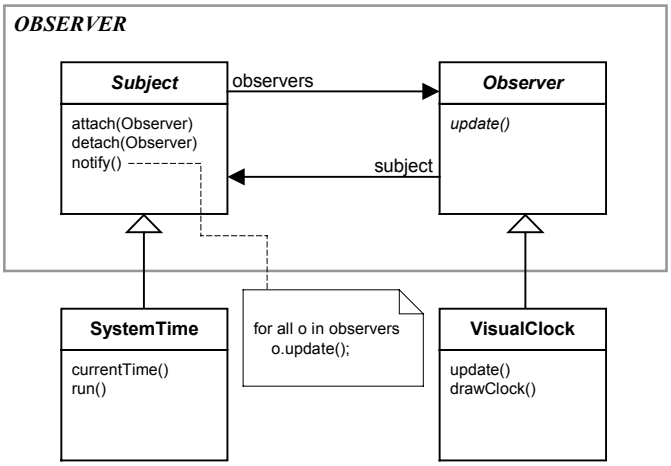
For å få en fleksibel struktur, bør man prøve å organisere det slik at det blir pekere og avhengigheter mellom færrest mulig av objektene, og spesielt slik at observerne er så uavhengig av hverandre som mulig. Om man har pekere som går på kryss og tvers i objektstrukturen, kan det være vanskelig å vedlikeholde og gjenbruke klassene andre steder.

Et sted hvor et slikt mønster med et dataobjekt og observere ofte forekommer, er vindusbaserte systemer, da informasjon ofte brukes mange steder, men med forskjellige visningsformer. Man kan for eksempel tenke på et regneark. Her kan datamengdene ofte bli presentert på forskjellige måter; gjennom selve regnearket i tabellform, ved diagrammer, grafer og så videre. De forskjellige visningsobjektene vil dermed ha behov for å få beskjed hver gang dataene endres, slik at de kan oppdatere seg.

Design-patternet *Observer* er en beskrivelse av hvordan slike avhengigheter kan struktureres og løses. *Observer* er et pattern i gruppa *"behavioral patterns"*, og er også kjent under navnet *publish-subscribe*.

5.2.1 Implementasjon av *Observer*-patternet

Den organiseringen som *Observer*-patternet foreslår, består av to abstrakte klasser, en klasse `Subject` som har eller kjenner de aktuelle dataene, og som skal gi beskjed til andre objekter som ønsker det hver gang det er forandringer. Klassen `Observer` skal være en superklasse for alle som ønsker å registrere seg som mottakere; altså å lytte til et `Subject`. Det er fullt mulig å la en `Observer` lytte til flere objekter av klassen `Subject`, men oftest er det bare aktuelt å lytte til ett. Et `Subject`-objekt vil derimot ofte ha knyttet til seg flere `Observere`-objekter.



Figur 5.1 Klassestruktur for *Observer*-patternet.

Som et eksempel på bruk av dette patternet, har jeg valgt å se på en klasse `SystemTime`, som jevnlig får et korrekt klokkeslett fra systemet, og som derved hele tiden sitter på riktig klokkeslett. Dermed kan denne klassen fungere som et `Subject`. Hver gang den har et nytt klokkeslett, skal den sende en melding til alle objekter som har registrert seg som interessenter, og disse tenker jeg meg kan være visuelle klokker som skal oppdatere seg for eksempel hvert sekund.

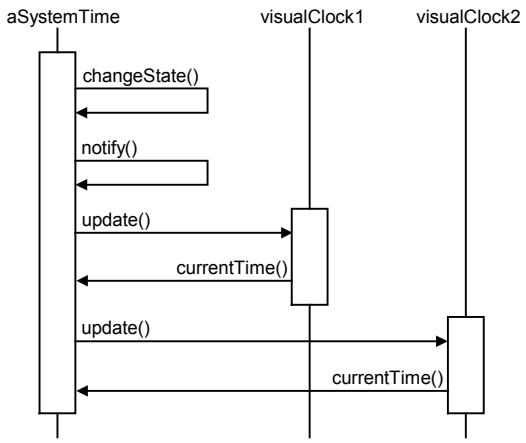
Klassestrukturen til hvordan [2] foreslår å implementere dette (med de abstrakte klassene `Subject` og `Observer`), samt hvordan mitt klokkeeksempel kan bruke det, er vist i Figur 5.1.

Et objekt av den aktuelle subklassen av `Subject` må ha en lokal liste over de objekter som ønsker å observere det respektive `Subject`-objektet. Metodene `attach` og `detach` brukes for å legge til og fjerne et `Observer`-objekt i denne lista. Når dataene i `Subject`-objektet endrer seg, blir metoden `noti-`

`fy` kalt, og objektet går da gjennom denne lista og kaller metoden `update` i alle registrerte `Observere`. I hvert av `Observer`-objektene henter så `update`-metoden selv informasjon fra det oppdaterte `Subject`-objektet. Dette skjer med et kall på en passelig metode i den aktuelle `Subject`-subklassen (her `currentTime` i klassen `SystemTime`).

Siden objektet selv skal hente de oppdaterte dataene, vil det være behov for en egen peker i hver aktuelle versjon av `Observer` til det tilknyttede `Subject`. Jeg har valgt å legge denne opp i superklassen `Observer`, selv om boka har lagt den i klassen for hver konkrete observer, i mitt tilfelle `VisualClock`.

Ved å legge pekeren inn i superklassen, slipper man å legger den inn i alle subklassene man oppretter. Denne skal jo være der for alle subklasser likevel. Men om man ikke har noen generisk mekanisme, vil da denne pekeren måtte ha en mer generell type, nemlig typen `Subject`, og det vil da ikke gå an å aksessere tidspunktet direkte uten først å legge inn en eksplisitt casting. Her vil dette forhåpentligvis løse seg ved å benytte en av de generiske mekanismene som jeg har presentert tidligere i oppgaven.



Figur 5.2 Eksempel på kallsekvens i *Observer*.

Hvis man ser på klassestrukturen i Figur 5.1, kan man legge merke til at denne har en del fellestrekk med eksempelet rundt grafer fra kapittel 4.5. Begge klassene som er tenkt programmert på forhånd (`Subject` og `Observer`), har gjensidige referanser til hverandre, og typene til disse referansene vil bli ”feil”, i den forstand at typene ikke stemmer med de konkrete objektene. Det må derfor legges inn eksplisitte castinger om man skal ha tak i metoder som ikke allerede er deklarert i superklassene.

Ved å utforme *Observer*-biblioteket som en generisk pakke, vil denne problematikken (som i graf-eksempelet) forsvinne. Ved hver konkret bruk av `Observer`-pakka, vil typene blir oppdatert i subklassene samtidig som funksjonaliteten arves som vanlig. Biblioteket kan da skisseres med følgende kode:

```
generic package observer virtual Subject, Observer;

abstract class Subject {
    // antar at det finnes en generisk versjon av Vector
    Vector<Observer> observers = new Vector<Observer>();

    void attach(Observer observer) {...}
    void detach(Observer observer) {...}

    void notify() {
        for (int i = 0; i < observers.size(); i++)
            observers.elementAt(i).update();
    }
}

abstract class Observer {
    Subject subject;
    ...
    abstract void update();
}
```

I brukerprogrammet som skal benytte *Observer*-patternet, definerer man at `VisualClock` skal erstatte klassen `Observer`, samtidig som `SystemTime` skal erstatte klassen `Subject`. Merk at man da får behov for at den klassen som leveres som parameter til `Vector` (altså `Observer`), også må kunne være virtuell til pakka. Ut fra definisjonen av generiske pakker ved tekstlig substitusjon, bør dette gå greit. Brukerprogrammet blir da som følger:

```
generic import observer with SystemTime, VisualClock;

class SystemTime (extends Subject) implements Runnable {
    Time currentTime() {...}

    // kjører klassen som en egen tråd, slik at oppdateringen
    // blir liggende i metoden 'run'.
    void run() {
        ...
        notify();
        ...
    }
}

class VisualClock (extends Observer) {
    Time time;

    void update() {
        // Siden typen til subject nå er oppdatert, kan
        // man hente ut currentTime uten bruk av casting.
        time = subject.currentTime();
        drawClock();
    }

    void drawClock() {...}
}

...
```

```
SystemTime st = new SystemTime();
VisualClock vc1 = new VisualClock();
VisualClock vc2 = new VisualClock();
st.attach(vc1);
st.attach(vc2);
```

Som man kan se i koden, har jeg valgt å legge klassen `SystemTime` ut i en egen tråd slik at den er helt uavhengig av hva som ellers skjer i de andre klassene. En slik implementasjon vil trolig være av interesse for mange implementasjoner av `Subject`, men slett ikke for alle. Jeg har derfor valgt å ikke legge den funksjonaliteten inn som en del av biblioteket.

Legg merke til at vi her ikke har sett noen nytte av at `Observer`-klassen er virtuell i pakka. Dette kan imidlertid komme til nytte dersom man i brukerprogrammet fra klassen `SystemTime` ønsker å kalle på metoder spesifikke for `VisualClock`.

Ved hjelp av generiske pakker, er det altså relativt enkelt å implementere *Observer* som et fleksibelt bibliotek, samtidig som man unngår problematikken rundt typer. *Observer* er i utgangspunktet ikke av de mest kompliserte designpatternene, og man hadde ikke her tapt så veldig mye ved å implementere eksempelet ved hjelp av det generiske idiom, men selvfølgelig må man da i subklassene legge inn eksplisitte castinger ved referanser som er plassert i biblioteket (noe vi ikke ønsker).

Om det i dette eksempelet er best å benytte parametriserte klasser eller generiske pakker, er kanskje en smakssak. Men Observer er liksom en *pakke* som enhet bestående av flere klasser, og det faller da mer naturlig å tenke i bane av generiske pakker. På slutten av kapittelet skal jeg forsøke å bruke flere av patternene samtidig i ett program, og da kan det komme inn flere argumenter rundt dette spørsmålet.

5.2.2 Variasjoner innenfor *Observer*-patternet

I boka [2] er, som sagt, *Observer*-patternet implementert slik at både `Subject` har en peker til klassen `Observer` og motsatt (selv om den der er plassert i subklassen). Hver gang et `Subject`-objekt endrer seg og `notify` blir kalt, går det først et `update`-kall til hver av objektene som er registrert som `observer`, og disse henter så den nye statusen ved å kalle metoder via pekeren til `Subject`.

En annen mulighet er at `update`-metoden selv tar den nye statusen, i mitt tilfellet klokkeslettet, som parameter. På denne måten skjer det bare ett metodekall i stedet for to, og det vil heller ikke være nødvendig med pekeren til `Subject`.

Hvis man velger denne løsningen, vil man altså ikke lenger ha behov for de gjensidige referansene, men man vil stå overfor et annet problem. Metoden `update` som ligger i den abstrakte klassen `Observer`, vil ikke ha definert

den korrekte parameteren (den vil jo variere fra bruk til bruk), og redeklarasjonen som skjer i `VisualClock`, vil derfor ikke bli oppfattet som en redefinisjon av den opprinnelige `update`, men vil i stedet bli oppfattet som en ny metode.

Dette problemet kan også løses med bruk av generiske pakker. Vi definerer da en ny, tom klasse i *Observer*-pakka som representerer parameteren (altså de dataene som skal leveres). Denne klassen skal også være virtuell slik at den kan erstattes ved pakkas bruk. Pekeren til `Subject` i klassen `Observer` vil nå være overflødig og kan dermed fjernes. Ellers vil eksempelet være likt som vist over.

Denne organiseringen med "pådytting" av data kan være ineffektiv hvis `Subject` inneholder mye data som skal formidles og forskjellige `Observere` bare er interessert i hver sine deler av dataene. I et slikt tilfelle er det mye overflødig data som sendes som parameter til `update`-metoden. Man er nesten nødt til å vurdere fra tilfelle til tilfelle hva som er mest lønnsomt, men for den videre utbygging av eksempelet, har jeg valgt å beholde `Subject`-pekeren som i boka, og lar dermed objektene selv hente de nødvendige dataene.

Av og til vil et `Observer`-objekt være registrert til å lytte på flere forskjellige `Subject`-objekter. Blant annet kan dette være tilfelle i et regneark hvor et diagram kan være avhengig av flere uavhengige datakilder. For å få til en mest mulig effektiv oppdatering, bør kun de endringene som er gjort bli tegnet opp på nytt. For å kunne oppnå dette, er det nødvendig for `Observer`-objektet å være klar over hvilket `Subject` som har kalt `update`.

En vanlig måte å gjøre dette på, er at kalleren av `update` sender med seg selv som parameter. Merk da at om ikke alle `Subject`-objekter er av samme klasse, så vil typingen av denne parameteren måtte være ganske generell, antakeligvis `Subject` selv. For å kunne hente dataene på riktig måte, må man da (inni `notify`) måtte teste på hvilket `Subject` som er kalleren, og så gjøre forskjellige castinger ut fra det. Et alternativ her ville være å ha én `notify`-metode for hvert `Subject` man er avhengig av. Da vil dette typeproblemet forsvinne.

En siste effektivisering jeg vil nevne i forbindelse med *Observer*-patternet, er å legge inn mulighet for å kunne spesifisere hvilke endringer man ønsker å få en notifikasjon om. En slik løsning kan man implementere ved å legge til en ny parameter i metoden for å registrere seg som observer. I tillegg til å sende med peker til seg selv, sendes også en spesifikasjon om hvilke endringer man er interessert i.

5.3 Composite

Det andre patternet jeg har valgt å gå nærmere inn på, er det såkalte *Composite*-patternet. Dette er et pattern som brukes for å strukturere objekter i objekttrær, hvor nodene i et tre enten kan være løvnoder eller roten i et nytt tre.

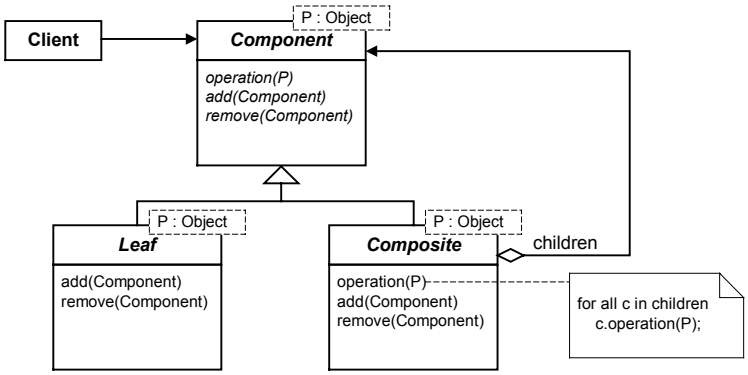
Tanken er at man utenfra, som ”klient”, skal kunne behandle hver node likt, uavhengig om det er en løvnode eller roten i et nytt tre. Når man skal gjøre noe på en slik trestruktur, vil det ofte gjøres som et rekursivt kall nedover i treet, og da vil ”klienten” for hver node være noden som står over den i treet.

Slike trestrukturer er veldig vanlige i mange typer anvendelser, og de kan uttrykkes ved at bladnodene representerer de primitive elementene i en struktur, og de indre nodene representerer forskjellige grupperinger av disse. *Composite*-patternet beskriver hvordan man kan opprette en slik struktur uten å forskjellsbehandle forskjellige typer noder.

5.3.1 Implementasjon av *Composite*-patternet

Nøkkelen til en slik implementasjon er å definere en abstrakt klasse `Component`, som representerer både bladnoder og indre noder. Alle objekter skal også ha en generell rekursiv metode, som abstrahert ut i fra alle kontekster kan kalles `operation`.

Superklassen som skal representere gruppene kalles `Composite`, og må i tillegg ha metoder for å kunne legge til og fjerne løvnoder og undergrupper. For å gjøre programmeringen mest mulig transparent, har jeg valgt å også definere grupperingsmetodene i klassen `Leaf`, men her inneholder de ingen funksjonalitet. De kan eventuelt defineres til å gi en feil. Jeg kommer tilbake til dette valget i avsnittet om varianter av patternet. Klassestrukturen for *Composite*-patternet er vist i Figur 5.3.



Figur 5.3 Generell klassestruktur for *Composite*-patternet.

Primitiver i et system som benytter *Composite*-patternet, skal her være definert som en subklasse av `Leaf`. Hver av disse klassene må implementere metoden `operation`, som inneholder kode for den spesifikke komponenten; for eksempel hvordan den skal tegnes opp.

Grupperinger skal være definert som subklasser av `Composite`, og her vil `operation` allerede (i selve `Composite`) være definert til å videreføre kallet til hver av primitivene eller undergruppene som en gruppering er bygget opp

av. I en subklasse av `Composite` kan man også redefinere `operation` til å utføre ekte operasjoner, men for å videreførekallet nedover i treet, må man også kalle versjonen som ligger i superklassen `Composite`.

Ved å benytte *Composite*-patternet, behøver klienter kun forholde seg til grensesnittet som er definert i `Component`. Hvis mottakeren av et metodekall er en subklasse av `Leaf`, blir kallet utført lokalt, men hvis mottakeren er en subklasse av `Composite`, blir metodekallet videreført til dens barn, men slik at det samtidig kan være mulig å utføre andre operasjoner før og/eller etter videresendingen.

Det medfører noen problemer når man på denne måten sprer funksjonalitet til flere delvis uavhengige komponenter, og et av problemene er at komponentene ofte er avhengig av samme data. Hvis man for eksempel skal tegne opp grafikk, er komponentene avhengig av å ha peker til grafikkobjektet det skal tegnes på.

En måte å løse dette problemet på, er å sende slike data med som en parameter til `operation`. Da vil man kunne få overført alle data i samme øyeblikk som det er behov for dem.

Siden vi ønsker å implementere *Composite*-patternet som et bibliotek, og siden disse dataene vil variere fra bruk til bruk, vil en naturlig måte å løse dette på være å pakke alle parametere inn i ett objekt, og typen til dette objektet bør kunne angis ved opprettelse av klassen. Med andre ord vil det være en god løsning å opprette `Component`-klassen generisk hvor parametertypen angis ved opprettelsen av en konkret brukerklasse. Klassene `Leaf` og `Composite` vil da arve denne generiske parameteren.

Biblioteket kan da implementeres som følger:

```
package composite;

abstract class Component<P> {
    abstract void operation(P param);
    abstract void add(Component comp);
    abstract void remove(Component comp);
}

abstract class Composite<P> extends Component<P> {
    // antar at det finnes en generisk versjon av Vector
    Vector<Component> children = new Vector<Component>();

    void add(Component comp) {...}
    void remove(Component comp) {...}

    void operation(P param) {
        for (int i = 0; i < children.size(); i++)
            children.elementAt(i).operation(param);
    }
}

 abstract class Leaf<P> extends Component<P> {
```

```
    // Definert som final slik at de ikke kan overstyres.
    // Inneholder ellers ingen funksjonalitet. De kan
    // eventuelt settes til å returnere en feilmelding.
    final void add(Component comp) {}
    final void remove(Component comp) {}
}
```

Som man kan se i koden over, er alle klassene definert abstrakte. Dette er gjort fordi man aldri skal instansiere objekter av dem. Det er beregnet at man alltid skal subklasse dem i et brukerprogram. Man kan også legge merke til at klassen `Leaf` ikke inneholder metoden `operation`. Denne *må* dermed deklareres i subklasser av `Leaf`.

Jeg har her valgt å implementere patternet ved hjelp av parametriserte klasser. Hvis man i stedet skal benytte generiske pakker, er nødt til å opprette en felles superklasse for sine brukerklasser som representerer primitivene. Tanken bak de generiske pakkene er at man skal instansiere en pakke ved å bytte ut hver av de virtuelle klassene med én egen brukerklasse. Det vil med andre ord være vanskelig å instansiere pakka på en slik måte at man får benyttet flere klasser samtidig.

En løsning på dette er, som sagt, at man internt i brukerprogrammet oppretter en felles superklasse for alle sine løvnoder, og denne felles superklassen må definere grensesnittet som skal være tilgjengelig fra en subklasse av `Composite`. Deretter kan man senere lage ”dynamiske” subklasser av den felles ”statiske” superklassen

Siden man ved generiske pakker er nødt til å opprette en felles superklasse, noe som ikke alltid vil være ønskelig, er det vel så bra å implementere patternet ved vanlig parametriserte typer; noe jeg tar utgangspunkt i videre i oppgaven.

Composite-patternet lar seg også implementere generelt ved hjelp av det generiske idiom, men da står man igjen med problemene rundt casting av variable. Alle steder hvor man opererer med parameteren som skal sendes med til `operation`, må man legge inn eksplisitte castinger.

I kapittel 3.6 skrev jeg litt om et potensielt ønske om å kunne ”renavne” metoder i subklasser. Det kan også være aktuelt i dette tilfelle da konkrete brukerklasser gjerne ikke opererer med metodenavn som `operation`. Jeg kommer tilbake til dette i eksempelet lenger ned.

5.3.2 Variasjoner innenfor *Composite*-patternet

Et av hovedpoengene med *Composite*-patternet, er at man utenfra sett skal behandle alle objekter på samme måte. For å få til dette på en god måte, må grensesnittet på objektene være like, og dette får man til ved å definere alle metoder som trengs i `Leaf` eller `Composite` i den abstrakte superklassen `Component`. Hvis man velger denne løsningen, er man sikker på at alle objekter kan aksessere alle metoder.

Det neste man må avgjøre, er hvor man ønsker å legge inn programlogikken. Dette blir en avveining mellom sikkerhet og transparens. Ved for eksempel å legge inn logikken til metodene `add` og `remove` i klassen `Component`, vil disse også være tilgjengelige i `Leaf`-klassen, selv om det ikke er naturlig at et primitiv skal kunne operere på gruppenivå. Dermed må det legges inn eksplisitt logikk som tester om et objekt er av klassen `Leaf`, og i så tilfelle skal det behandles spesielt.

Denne løsningen har ikke jeg brukt. I mitt eksempel blir selve funksjonaliteten først definert i klassen `Composite`. Ved å gjøre det på denne måten, trenger jeg ikke ta hensyn til løvnodene. Likevel vil grensesnittet til metodene være tilgjengelig i klassen og subklassene til `Leaf`, noe som også kan være et sikkerhetshull. Dette har jeg løst ved å la `Leaf` overstyre metodene rundt gruppering, hvor hver metode blir definert som `final` samtidig som de er tomme (eller gir en feilmelding). De vil dermed ikke kunne redefineres i eventuelle subklasser.

En annen ting som kan varieres i *Composite*-patternet, er om det skal opereres med en peker til forelderkomponenten. Ved å legge inn dette, vil det bli enklere å traversere et objekttre begge veier, og det vil blant annet bli enklere å fjerne objekter fra treet. Siden dette ikke vil gjøre eksempelet noe mer komplisert med tanke på typer (foreldrepekeren skal være typet med `Composite`), har jeg utelatt denne delen fra implementasjonen.

I flere tilfeller, særlig ved opptegning av objekter, er det viktig at komponentene blir traversert i en bestemt rekkefølge. Man kan legge inn funksjonalitet for dette i design-patternet, men i eksempelet under bruker jeg rett og slett innsettingsrekkefølgen som utgangspunkt. Det er derfor viktig at komponenter blir lagt inn i den rekkefølgen de skal behandles.

Hvis man skulle legge inn støtte for en spesiell rekkefølge, måtte man i innsettingsmetoden også kunne spesifisere plasseringen for eksempel ved en slags ”prioritet”. Denne utvidelsen vil heller ikke gjøre eksempelet rundt typer vanskeligere, så jeg har utelatt denne delen også.

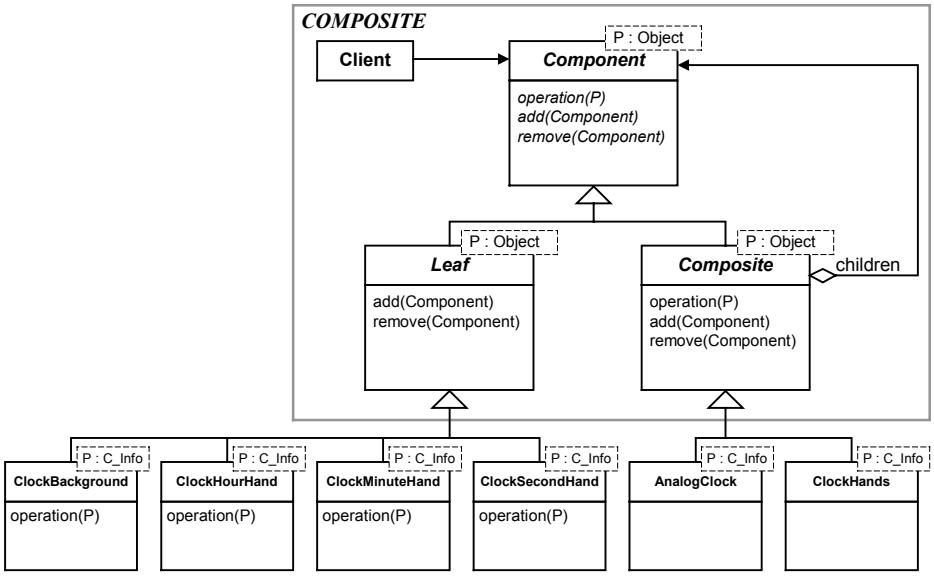
I boka inneholder klassene en grupperingsmetode med navn `getChild`. Dette er en metode som blant annet gjør det enklere å traversere et tre på andre måter. Nå traverseres de kun fra start til slutt, men det er nok for mine eksempler.

5.3.3 Eksempel på bruk av *Composite*-patternet

Som et eksempel har jeg brukt *Composite*-patternet i forbindelsen med opptegning av en analog klokke (mens en digital klokke blir representert enklere). Jeg har jeg delt klokka opp i forskjellige deler; urskive og de forskjellige viserne. Hele klokka vil da naturlig være representert som et objekt av subklassen `AnalogClock` av `Composite`. Jeg har så valgt å representere viserne som objekter av hver sin klasse, samtidig som jeg også har en klasse som samler alle viserne. Dette kan virke noe tungvint, men jeg har valgt denne

strukturen for å få demonstrert bruk av design-patternet. Klassestrukturen jeg benytter i dette eksempelet er vist i Figur 5.4, og et eksempel på en objekt-struktur er vist i Figur 5.5.

Objekter av klassene `AnalogClock` og `ClockHands` holder stort sett kun på et sett delobjekter. Med andre ord inneholder de selv ikke noe særlig funksjonalitet, men bare pekere til objekter som selv utfører opptegningen. Disse klassene arver `operation` fra klassen `Composite`.



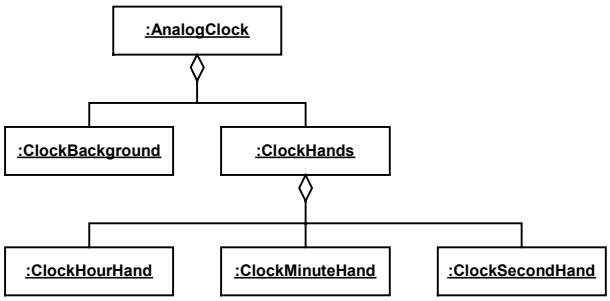
Figur 5.4 Klassestruktur for *Composite*-patternet sammen med en klokke.

Eksempelvis har hvert objekt av klassen `ClockHands` under seg et objekt for hver viser, og disse blir laget av konstruktøren til klassen. I denne klassen har jeg også vist hvordan man i en subklasse til `Component` kan sette strengere krav til den generiske parameteren. Her antar jeg at parameteren må være en klasse av typen `ClockInfo` (forkortet `C_Info` på figuren) som inneholder for eksempel peker til korrekt grafikkelement og klokkeslettet. Klassen kan da for eksempel bli:

```
import composite;

class ClockHands<P extends ClockInfo> extends Composite<P> {
    public ClockHands() {
        super();
        add(new ClockHourHand<P>());
        add(new ClockMinuteHand<P>());
        add(new ClockSecondHand<P>());
    }
}
```

I dette eksempelet blir bindingen av delobjektene helt statisk, men ved å legge inn metoder som kan endre objektene etter at instansieringen er utført, kan man enkelt bytte ut en eller flere av de aggregerte delobjektene dynamisk. Med andre ord kan man forandre utseende på klokka mens programmet kjører, ved for eksempel å ta bort sekundviseren.

Som et eksempel på et primitiv, har jeg valgt å vise hvordan man kan implementere klassen `ClockHourHand` som representerer timeviseren i klokka. Denne klassen trenger i utgangspunktet ikke å implementere mer enn metoden `operation`, og i denne skal det utføres opptegning av viseren.



Figur 5.5 Objektstruktur for samme eksempel som i Figur 5.4.

Man kan tenke seg at parameterobjektet som følger med, inneholder peker til grafikkobjektet det skal tegnes på; i tillegg til informasjon om størrelse, plassering og ikke minst klokkeslettet. Klassen kan da bli omtrent slik:

```
class ClockHourHand<P extends ClockInfo> extends Leaf<P> {
    void operation(P param) {
        // Henter ut grafikkobjektet det skal tegnes på.
        Graphics g = param.graphics();
        // Utfører de korrekte tegnekommandoer.
        ...
    }
}
```

Som jeg nevnte tidligere i kapittelet, kan det være ukurant å måtte operere med metodenavn som `operation` i de aktuelle anvendelser, bare fordi biblioteket skal være så generelt som mulig. Tidligere presenterte jeg tanker om at det hadde vært kjekt (og i mange situasjoner nødvendig) med en mekanisme som gjorde det mulig å angi at en metode i en subklasse skulle ha et annet navn, men samtidig bli en redefinisjon.

I dette tilfelle ville det kanskje vært mer naturlig om tegnemetoden i `ClockHourHand` het `draw` i stedet for `operation`. I så fall kunne man med mekanismen jeg foreslo i kapittel 3.6, oppnå denne navneendringen med følgende syntaks:

```
class ClockHourHand<P extends ClockInfo>
    extends Leaf<P> <where operation -> draw> {
```

```
    void draw(P param) {
        // Innholdet er det samme som over.
        ...
    }
}
```

I denne versjonen av klassen vil altså metoden `draw` være en redefinisjon av metoden `operation` fra klassen `Component`. Som jeg nevnte tidligere, er ikke syntaksen for navneendring nødvendigvis god, da den blander seg med de generiske typene, men den viser i hvert fall hva jeg kunne ønske var mulig.

5.4 Bridge

Når man programmerer med klasser og subklasser, vil man kunne oppleve at forskjellige logiske hierarkier lett kan filtrere seg inn i hverandre, slik at man får en klassestruktur som er veldig komplisert. *Bridge*-patternet foreslår en løsning på dette hvor man ved hjelp av en aggregeringspeker, enklere kan holde klassehierarkiene atskilt slik at de kan utvikles og benyttes uavhengig av hverandre.

5.4.1 Motivasjon ved hjelp av et eksempel

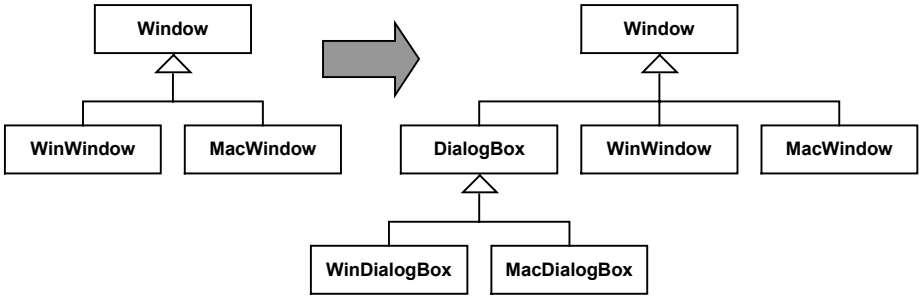
Når man designer et program, vil man ofte oppdage at for en og samme logiske ”grunnklasse”, kan man lage subklasser av denne ut fra forskjellige filosofier. Eksempelvis kan man se på utvikling av programmer som skal støtte forskjellige vindusplattformer. Om man tar utgangspunkt i en klasse `Window` som skal representere et ”vindu” på skjermen, kan man for eksempel lage subklasser av denne på følgende to måter:

- For det første har man forskjellige typer vinduer, for eksempel dialogbokser og modale vinduer, og disse kan naturlig representeres med subklasser av `Window`.
- I tillegg skal man håndtere de forskjellige plattformene som vindussystemet skal implementeres på, og for å få til dette, kan man vurdere å lage en subklasse av `Window` for hver plattform, for eksempel `WinWindow`, `MacWindow` osv.

Om man prøver å løse begge disse problemene ved å subklasse ”grunnklassen” `Window`, blir det raskt en veldig komplisert struktur, og det kan også være uklart hvordan strukturen i det hele tatt skal være. Figur 5.6 viser hvordan dette klassehierarkiet kunne utvikle seg.

I tillegg til at strukturen er komplisert, vil det samtidig være vanskeligere å abstrahere selve applikasjonen bort fra en konkret implementasjon for et vindussystem, noe som ikke er heldig. Programkoden vil raskt blir plattformavhengig ved at man må instansiere et objekt av en konkret klasse som har en spesifikk plattformimplementasjon, når man skal opprette et vindu. Opprettes det for eksempel et objekt av klassen `MacWindow`, bindes vinduspekeren opp mot den spesifikke Macintosh-implementasjonen, og resultatet er at det der-

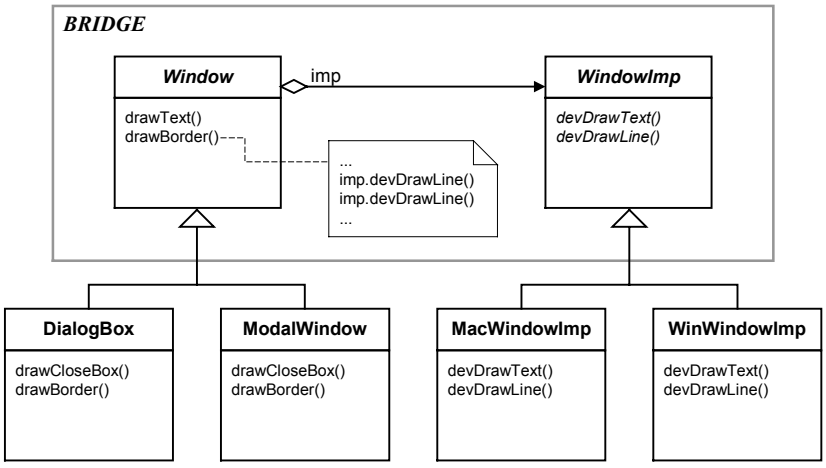
med blir vanskeligere å konvertere applikasjonen til en annen plattform senere.



Figur 5.6 Klassestruktur for vindussystemer med kombinert klassehierarki.

Bridge-patternet hjelper til med å løse disse problemene ved å skille klassehierarkiene, men slik at de kobles sammen gjennom en aggregeringspeker; gjerne kalt `imp`. Aggregering er en mer dynamisk form for binding, så denne kan til og med (hvis det er lagt opp til det) byttes mens programmet eksekverer.

I Figur 5.7 kan man se ideen rundt *Bridge*-patternet. Tankegangen er at klassen `Window` og alle subklassene er implementert med kode som forholder seg til operasjoner som tilbys i grensesnittet til `WindowImp` og som implementeres i de forskjellige plattformklassene. Dette frigjør vindusklassene fra den plattformspesifikke implementasjonen.



Figur 5.7 Klassestruktur for vindussystemer med bruk av aggregering.

Forbindelsen mellom `Window` og `WindowImp` kalles gjerne en bro; derav navnet *Bridge*-pattern. En slik binding vil gjøre det enklere å bruke, bytte ut og videreutvikle klassehierarkiene uavhengig av hverandre.

I flere nyere programmer og operativsystemer, er det mulig å spesialsy eller velge forskjellig utseende. Dette gjøres som regel gjennom bruk av såkalte *temaer* eller *skins*. I disse programmene og operativsystemene kan man velge forskjellige skins fra en liste, og deretter forandrer hele utseende til applikasjonen seg. Slik funksjonalitet kan "lett" implementeres ved hjelp av *Bridge*-patternet, ved å la `imp`-pekeren bytte fra én konkret implementasjon til en annen.

5.4.2 Generell bruk av *Bridge*-patternet

Jeg har over benyttet et konkret eksempel når jeg har forklart hvordan *Bridge*-patternet fungerer. Det er derfor viktig å poengtere at dette bare er ett eksempel og at det finnes mange andre problemstillinger som kan løses på en tilsvarende måte. Forfatterne av [2] har satt opp en punktliste over tilfeller hvor det kan være hensiktmessig å benytte design-patternet. De viktigste punktene er at det passer å bruke *Bridge*-patternet i følgende tilfeller:

- Hvis man ønsker å forhindre en permanent binding mellom abstraksjonen og dens implementasjon. For eksempel hvis man ønsker at implementasjonen skal velges eller byttes ved programeksekvering.
- Hvis både abstraksjonen og dens implementasjon skal kunne utvides med subklasser. I så tilfelle hjelper *Bridge*-patternet med å gjøre disse uavhengige av hverandre.
- Hvis man ønsker å skjule konkrete implementasjoner for klienten. Ved hjelp av *Bridge*-patternet, kan en programmerer kun få lov til å forholde seg til et grensesnittet av de konkrete implementasjonene.
- Hvis man ønsker å benytte en konkret implementasjon sammen med flere forskjellige abstraksjonsobjekter, og her har man da også, som sagt, mulighet til å skjule kode for klienten.
- Hvis man ønsker å kunne jobbe med abstraksjonen og implementasjonen uavhengig av hverandre. Hvis man gjør en endring i for eksempel implementasjonen, skal man slippe å endre og kompilere abstraksjonen.

Hvis man vet at man bare vil operere med én implementasjon, vil det ikke være nødvendig å lage en abstrakt superklasse for implementasjonen (her `WindowImp`), men det kan likevel være en fordel å operere med en `imp`-peker og et konkrete implementasjonsobjekt. Det kan kanskje virke dumt å benytte *Bridge*-patternet ved bare én konkret implementasjon, men det kan være hensiktsmessig hvis man vil unngå at endring av implementasjonen ikke påvirke abstraksjonen.

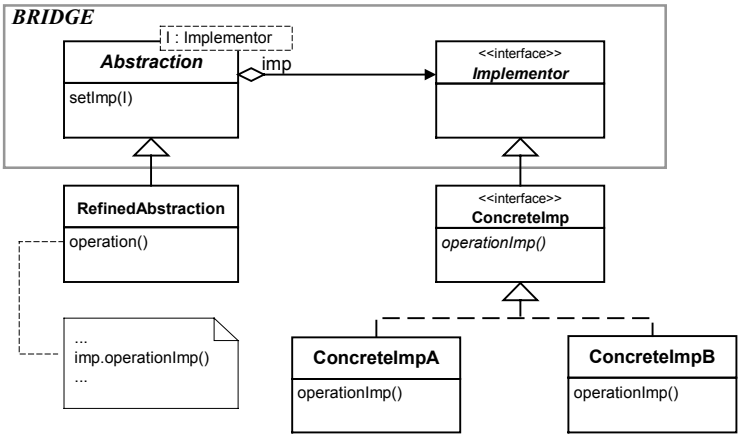
5.4.3 Implementasjon av *Bridge*-patternet som bibliotekskklasser

I vinduseksempelet var *Bridge*-patternet integrert sammen med klassene for et spesielt eksempel. Hvis man skal lage *Bridge*-patternet så abstrakt at det skal

kunne legges inn i et bibliotek for deretter å bli benyttet i forskjellige sammenhenger, er man nødt til å finne generelle klassenavn, og det er vanlig å kalle klassene `Abstraction` og `Implementor`.

Siden klassene som ligger i biblioteket er helt generelle, inneholder de heller ingen metoder bortsett fra metodene for å tilordne en ny `Implementor`. Alle andre metoder vil naturlig ligge i subklasser til klassene i selve *Bridge*-patternet. Med andre ord får man bare implementert "strukturen" i biblioteket. Alt annet vil være spesifikt for den aktuelle bruken.

I klassen `Abstraction` er det en referanse til den aktuelle `Implementor`. I et brukerprogram hvor man benytter subklasser, vil denne referansen få en for generell type, slik at man må legge inn castinger. Derfor er man avhengig av at man implementerer biblioteket ved hjelp av en generisk mekanisme.



Figur 5.8 Klassestruktur for *Bridge*-pattern.

Klassene som utgjør *Bridge*-patternet, lar seg greit implementere både ved hjelp av parametriserte klasser og generiske pakker. Siden abstraksjonen i dette patternet skal jobbe mot et kjent grensesnitt for implementasjonen, vil man automatisk ha en felles superklasse for implementasjonene. Dermed har man ikke de samme problemene jeg kort beskrev for *Composite*-patternet.

Jeg har likevel, blant annet for det kombinerte eksempelets skyld, valgt å skrive biblioteket ved hjelp av parametriserte typer, og klassen `Abstraction` må dermed ha en generisk parameter slik at man kan spesifisere den konkrete implementasjonen når man skal benytte patternet. Se klassestrukturen for patternet i Figur 5.8.

Et eksempel på bibliotekskласser kan da være som følger:

```
package bridge;

abstract interface Implementor {
    // Dette interfacet inneholder ingenting, men benyttes
    // som superinterface for alle konkrete implementasjoner.
```

```
}

abstract class Abstraction<I implements Implementor> {
    I imp;

    setImp(I newImplementor) {
        imp = newImplementor;
    }
}
```

I et brukerprogram som benytter patternet, må man så definere klassen for den konkrete abstraksjonen som en subklasse av `Abstraction`. Et eksempel på en aktuell bruk kan være:

```
import bridge;

class RefinedAbstraction extends Abstraction<ConcreteImp> {
    ...
}

// Deklarasjon av eventuelle subklasser av Refinded-
// Abstraction.
...

interface ConcreteImp extends Implementor {
    void operationImp();
}

// Deklarasjon av diverse konkrete implementasjoner.
...
```

Siden brukerklassene her må defineres til å være subklasser av bibliotekskla-ssene, vil det være vanskelig å benytte klasser som er definert tidligere i andre sammenhenger. Dette vil være et problem enten om biblioteket skrives ved hjelp av parametriserte klasser eller generiske pakker. Likevel vil problemet kanskje ikke være så veldig aktuelt siden klassene *må* defineres med tanke på patternet. Derfor vil man nok i de aller fleste tilfeller ha skrevet klassene selv.

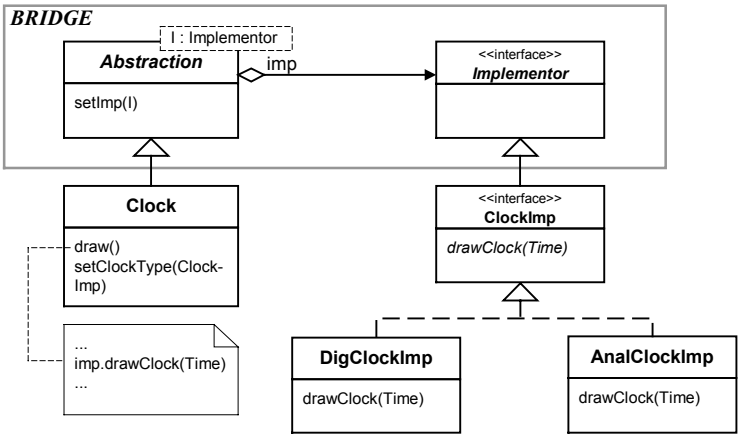
5.4.4 Eksempel med bruk av *Bridge*-patternet

Jeg viser her et eksempel hvor jeg benytter *Bridge*-patternet i forbindelse med opptegning av en klokke på skjermen. Det skal være mulig å velge om det skal tegnes opp en digital eller analog klokke. Siden disse skal kunne skiftes dynamisk, har jeg valgt å benytte *Bridge*-patternet.

Her vil klassen `Clock` være en utvidet abstraksjon, mens jeg har valgt å opprette klassene `DigClockImp` og `AnalClockImp` som begge ligger under det felles interfacet `ClockImp`. Det felles interfacet sørger for at begge subklassene inneholder metoden for å tegne opp klokka. Se klassestruktur i Figur 5.9.

I brukerprogrammet som henter inn *Bridge*-patternet fra biblioteket, bør man angi sin egen konkrete `Implementor` som generiske parameter, i dette tilfellet `ClockImp`. På denne måten kan man jobbe som om biblioteket var spesi-

alskrevet for de aktuelle typene, og det vil da være mulig å aksessere metoder direkte.



Figur 5.9 Klassestruktur for *Bridge*-pattern brukt med klokker.

```
import bridge;

class Clock extends Abstraction<ClockImp> {
    void draw() {
        // Sender beskjed om å tegne opp klokken til den
        // aktuelle implementasjonen som er linket inn.
        imp.drawClock(Time);
    }

    void setClockType(ClockImp type) {...}
    ...
}

interface ClockImp extends Implementor {
    void drawClock(Time time);
}

class DigClockImp implements ClockImp {
    void drawClock(Time time) {...}
}

class AnalClockImp implements ClockImp {
    void drawClock(Time time) {...}
}
```

Ved å definere klokka slik jeg har skissert over, får jeg en dynamisk og fleksibel løsning. Når man skal benytte klokka i et program, kan brukerne ved hjelp av et ”flagg”, velge om det skal tegnes en analog eller digital variant. Hvis man for eksempel ønsker å skifte fra en digital til en analog klokke, trenger man bare å endre `imp`-pekeren til å peke på et objekt som representerer den analoge klokka. Dette kan man for eksempel gjøre med å implementere en metode lignende `setClockType` som jeg har skissert i over. Neste

gang klokka tegnes opp etter at denne kommandoen er kjørt, vil den bytte fra å være analog til å bli digital.

5.5 Kombinering av design-patterns

Til nå i dette kapittelet har jeg sett på tre ulike design-patterns, og forsøkt å implementere disse generelt slik at de kan legges ut som bibliotekspakker. Dette har fungert relativt greit, selv om alle patternene bærer preg av å bli nokså enkle rammeverk, og det kan vel diskuteres hvilken nytte det vil ha å legge dem ut som bibliotheksklasser.

De har imidlertid vært nyttige i forbindelse med å se hvordan typeproblemer kan opptre i forholdet mellom det å "skille ut" ting som bibliotheksklasser, for senere å skulle tilpasse dem til konkrete situasjoner. Jeg har altså, som jeg har vært inne på, litt tilfeldig valgt å implementere *Observer* med generiske pakker og *Composite* og *Bridge* med generiske klasser.

Da jeg presenterte hvert pattern, tok jeg samtidig utgangspunkt i et eksempel som omhandlet klokker. Jeg vil i dette avsnittet prøve og kombinere disse eksemplene ved å ta utgangspunkt i `swing`-biblioteket som følger med Java 1.2, og lage et program som kan opprette vinduer som hver inneholder en klokke. Hver klokke kan representeres ved et objekt av en ny klasse `JClock` som henter inn og utnytter funksjonalitet fra alle de tre design-patternene jeg har gått gjennom, samtidig som den henter vindusfunksjonalitet fra `Swing`-biblioteket.

5.5.1 Beskrivelse av `Swing`-biblioteket og klassen `JClock`

`Swing`-biblioteket, som først ble presentert på JavaOne-konferansen[3] i 1997 og som erstatter `awt`-biblioteket, er beregnet for å jobbe med grafikk i Java. `Swing`-biblioteket inneholder mange klasser som hver representerer forskjellige grafikkelementer for vindusbaserte applikasjoner. Alle disse har navn som starter med bokstaven *'J'* (`JMenu`, `JButton` osv.), og felles for dem er at alle er subklasse av en felles abstrakt klasse `JComponent`.

Den nye klassen, kalt `JClock`, skal kunne brukes på lik linje med andre `swing`-komponenter, og den skal ha to varianter (subklasser) som kan vise tiden digitalt eller analogt. Klokkas størrelse på skjermen skal også kunne endre seg, og den skal bruke, så godt som det er mulig, den plassen den er blitt tildelt (følge vindusstørrelsen).

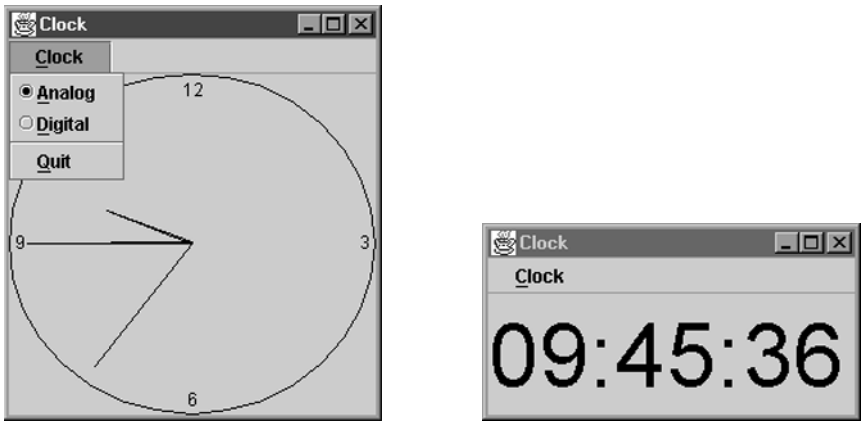
Siden den nye klassen skal kunne spille sammen med `swing`-biblioteket, er `JClock` nødt til å være en ekte subklasse til `JComponent`. Siden klassen i tillegg skal arve funksjonalitet fra de design-patternene jeg har presentert, vil det melde seg et behov for multippel arv; i hvert fall hvis man skal kunne implementere klassen slik det i utgangspunktet kanskje ville være naturlig.

[3] Offisiell side for JavaOne finnes på url: *http://java.sun.com/javaone/*.

Jeg vil derfor her anta at Java tillater multippel arv omtrent som i C++, men skal senere gi noen vurderinger av i hvilken grad man i stedet kunne klare seg med statisk multippel arv slik det er definert for generiske pakker.

For å vise hvordan den nye komponenten kan brukes, har jeg også laget et program (klassen `Program`) som åpner to vinduer som hver viser en klokke. De to klokkene benytter det samme `SystemTime`-objektet. Med andre ord "lytter" de som en del av *Observer*-patternet, til samme `Subject`-objekt. Klokkene vil altså vise akkurat samme tid, og de har hver ansvar for å tegne seg selv opp.

Ved hjelp av en meny knyttet til selve klokkevinduet, kan man endre en klokke fra å være analog til å bli digital og omvendt. Denne funksjonaliteten oppnås ved å benytte *Bridge*-patternet. Ved opptegning av den analoge versjonen, har jeg splittet klokka opp i flere subkomponenter som visere og bakgrunn, og disse bindes altså sammen ved hjelp av *Composite*-patternet. I Figur 5.10 er det et bilde fra en skjermdump av vinduene tegnet av en versjon av programmet som er implementert ved hjelp av det generiske idiom.



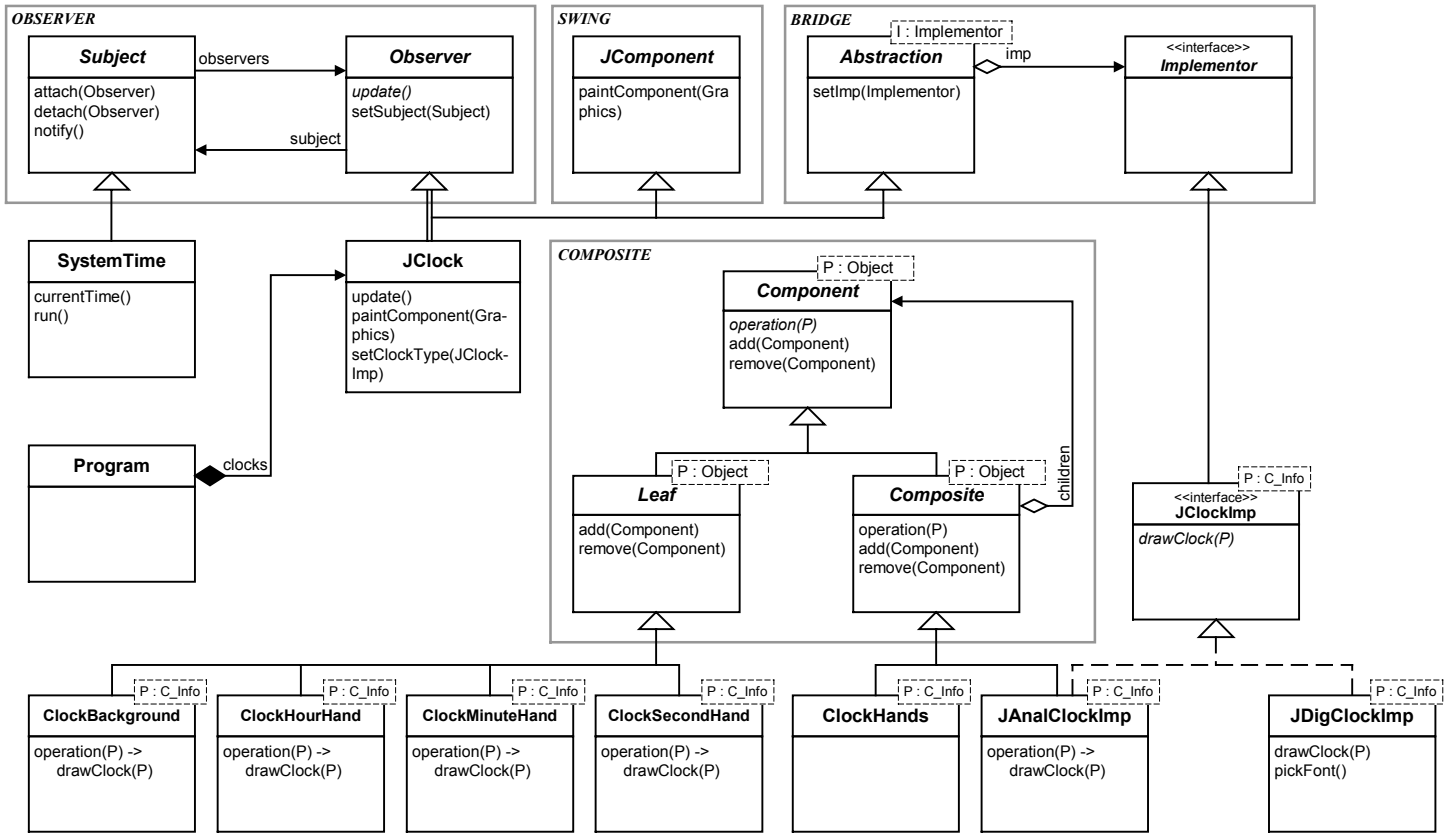
Figur 5.10 Skjermdump fra klokkeeksempelet.

I neste delkapittel vil jeg presentere hvordan jeg ser for meg den nye komponenten implementert, og jeg vil her ta utgangspunkt i de generiske mekanismene jeg har omtalt tidligere i oppgaven og de implementasjonene jeg har foreslått av hvert pattern tidligere i dette kapittelet.

For sammenligningens skyld, har jeg også implementert, og dermed fått testet ut, et tilsvarende program i nåværende versjon av Java ved hjelp av det generiske idiom. Men siden jeg her ikke kan benytte multippel arv, vil noe av strukturen og koden være annerledes. I appendikset bak i oppgaven er dette programmet angitt i sin helhet.

5.5.2 Implementasjon ved hjelp av nye mekanismer

Komplett klassestruktur for eksempelet som inkluderer `JClock`, er vist i Figur 5.11. Her er det forutsatt at språket har generiske mekanismer og multippel arv. Klassestrukturen er nok på mange måter kunstig innviklet og stor, men den er satt opp mer som en studie i programstruktur enn for at et program av `JClock` sin størrelse, skal bli implementert på greiest mulig måte. Man kan se det som et eksempel på hvordan en struktur typisk kan være når man skal hente funksjonalitet fra flere bibliotekspakker, og få det til å fungere sammen.



Figur 5.11 Komplett klassestruktur for klokkeeksempelet.

Hovedklassen for den nye komponenten er `JClock`, og jeg har altså satt inn en brukerklasse `Program` som oppretter ett eller flere grafiske vinduer hvor `JClock`-komponenten vil fylle selve vinduet.

`JClock` må altså være en ekte subklasse av `JComponent`, men den skal samtidig være mottaker av "systemklokka", og er dermed en statisk subklasse av `Observer` fra *Observer*-patternet. Til slutt, siden klokka dynamisk skal kunne skifte fra å være digital til å bli analog, er den også en subklasse av `Abstraction` fra *Bridge*-patternet.

Som tidligere nevnt, brukes også *Composite*-patternet til å administrere opptegningen av den analoge klokka. Som man kan se i klassestrukturen, arver `JClock` dermed fra tre forskjellige klasser, hvorav klassen arver ”statisk” (gjennom generiske pakker) fra `Observer`.

Før jeg går nærmere inn på konkrete utfordringer rundt implementasjonen, vil jeg kort beskrive hvordan komponenten fungerer. Siden `JClock` er en subklasse av `JComponent`, må klassen definere metoden `paintComponent`. Denne metoden vil automatisk kalles når et vindu er nødt til å tegnes opp på nytt, for eksempel fordi et vindu igjen blir synlig på skjermen.

Siden `JClock` også skal være mottaker for systemklokka (som en del av *Observer*-patternet), må klassen redefinere metoden `update`. Denne metoden vil hente det oppdaterte klokkeslettet for så å kalle `paintComponent`.

Metoden `paintComponent` vil være ganske kort. Den kaller rett og slett samme metode i superklassen, for at bakgrunn og lignende skal tegnes opp, før den (som en del av *Bridge*-patternet) kaller metoden `drawClock` i den innlinkede klokkeimplementasjonen som utfører selve opptegningen av klokka. Metoden `drawClock` tar en parameter av typen `ClockInfo` hvor all informasjon rundt det grafiske elementet og klokkeslettet ligger.

Hvis implementasjonen i *Bridge*-patternet angir en digital klokke, tegnes denne opp direkte, men hvis det er en analog klokke, vil opptegningen bli delegert ut til subkomponentene gjennom *Composite*-patternet. Hele veien videresendes parameterobjektet slik at de forskjellige objektene kan utføre opptegning uavhengig av hverandre.

Ut i fra klassestrukturen jeg har skissert over, vil `JClock` kunne bli implementert som følger:

```
generic import observer with SystemTime, JClock;
import swing;
import bridge;
import composite;

class JClock (extends Observer)
    extends JComponent, Abstraction<JClockImp> {
    /* Variabel som holder på selve klokkeslettet. */
    Time klokkeslett;

    void update() {
        /* Henter inn det nye klokkeslettet. */
        klokkeslett = subject.currentTime();
        /* Magisk kall for å trigge paintComponent. */
        repaint(new Rectangle(getWidth(), getHeight()));
    }

    void paintGraphics(Graphics g) {
        /* Oppdaterer andre deler av vinduet. */
        super.paintCompoent(g);
        /* Oppretter et objekt som inneholder all nødvendig */
        /* info. rundt klokka og det grafiske vinduet.      */
```

```
        ClockInfo clockInfo = new ClockInfo(...);
        /* Sender et kall om at klokka skal tegne seg opp. */
        imp.drawClock(clockInfo);
    }

    ...
}

// Her vil de andre klassene bli deklarert fortløpende...
```

Som man ser i deklarasjonen av `JClock`, blir klassen ganske kort siden den arver det meste av funksjonaliteten fra superklassene sine. Man kan også legge merke til at den foreslåtte syntaksen for generiske pakker, kolliderer noe med syntaksen for vanlig subklassing. Det kan virke ganske kunstig at én av superklassene settes opp i parentes, mens de andre står oppført på "vanlig" måte. Hovedvekt i denne oppgaven er imidlertid ikke lagt på syntaksen, så jeg går ikke nærmere inn på dette.

I kapittel 4 diskuterte jeg kort hvorvidt innføring av statisk multippel arv vil være tilstrekkelig, med tanke på at en slik mekanisme er noe enklere å implementere enn full multippel arv, og at forutsetningen om å innføre full multippel arv i Java kanskje kan være noe urealistisk. Jeg nevnte også at statisk multippel arv i utgangspunktet kan komme til kort i visse situasjoner. Jeg vil nå gå litt dypere inn på dette siden jeg har et større konkret eksempel å diskutere ut fra.

Den kritiske klassen i eksempelet over er `JClock`, siden den arver fra tre forskjellige klasser, hvorav bare én arves statisk. Hvis man skal kunne klare seg med bare statisk multippel arv, bør klasser som arver multippelt, bare bruke statisk arv. Spørsmålet her blir da om det ville være mulig å endre strukturen slik at `JClock` kunne arve statisk fra både `swing` og *Observer* også.

For at arven fra `JComponent` skulle kunne være statisk, måtte den delen av `swing`-biblioteket som inneholdt denne klassen, være implementert som en generiske pakke hvor `JComponent` var deklarert som virtuell. Denne ideen får imidlertid fram en av de svake sidene ved de generiske pakkene, nemlig at en virtuell klasse ikke får lov å ha definert andre subklasser inne i pakka. Dette var et av kravene som jeg satte i kapittel 4, fordi slike subklasser vil få en annen superklasse når den generiske pakka instansieres og den virtuelle klassen blir erstattet. Subklasser av `JComponent` vil imidlertid være en naturlig del av en slik generisk pakke.

Konklusjonen er dermed at det ikke er kurant å la `JComponent` være en virtuell klasse i en generiske pakke, og det vil derfor neppe være mulig å arve fra denne statisk.

Hva så med arven fra klassen `Abstraction` i *Bridge*-patternet? Som sagt tidligere, var det litt tilfeldig at *Bridge*-patternet ikke ble laget som en generisk pakke, og vi kan derfor tenke oss at det faktisk ble gjort slik, da naturlig med interfacet `Implementor` deklarert virtuelt. For mitt formål måtte imid-

lertid også `Abstraction` være virtuell, men det burde ikke skape noen nye problemer.

Med andre ord står jeg igjen med en klasse `JClock` som arver statisk fra to klasser og "dynamisk" fra en tredje. I dette tilfellet ville man dermed bare ha enkel dynamisk arv, og derfor vil det kanskje kunne gå greit med kun statisk multippel arv, men generelt vil det *ikke* holde. Hvis man for eksempel ønsker å benytte flere biblioteker som `swing`, vil statisk multippel arv raskt bli en for "svak" mekanisme for å kunne løse problemene som dukker opp.

Som jeg har skissert, vil generiske pakker stort sett være godt egnet når man har en pakke som skal hentes inn én gang samtidig som de virtuelle klassene hver skal erstattes med én "subklasse". Ved `swing`-biblioteket er ikke dette tilfellet, og det egner seg dermed dårlig for implementasjon ved hjelp av statiske pakker.

Det synes å være et interessant prosjekt å gå dypere inn på dette området. Man burde da gå inn på flere eksempler og undersøke om man kunne definere generiske pakker på en bedre måte som kanskje også kunne løse noen av problemene jeg har kommet over. Jeg har imidlertid valgt å la dette spørsmålet stå åpent slik at andre eventuelt kan ta tak i det. Noe av grunnen for dette, er som sagt at fokus i denne oppgave er generiske typer og ikke multippel arv.

5.5.3 Implementasjon ved hjelp av det generiske idiom

Siden de mekanismene jeg tenker meg brukt i programmet over, ikke er implementert i Java, har jeg også laget en versjon av programmet i standard Java (versjon 1.2), og dette er gjengitt i appendikset bak i oppgaven.

Strukturen for denne implementasjonen er ganske lik den i Figur 5.11, men visse elementer, blant annet klassen `JClock`, må løses annerledes. Det vil ikke lenger være mulig å la én enkelt klasse arve funksjonaliteten på samme måte, så man er nødt til å dele klassen opp i flere enheter som fungerer sammen.

I mitt forslag har jeg latt `JClock` være en subklasse til `JComponent`, noe den må være for å ha vindusfunksjonalitet, mens jeg har opprettet en annen klasse `ClockWindow` som arver fra `Observer` samtidig som den inneholder en peker til `JClock`. Videre finnes det ingen enkel, naturlig måte å integrere *Bridge*-patternet på slik som over, så jeg har derfor lagt `imp`-pekeren rett i klassen `JClock`. Med andre ord har jeg fjernet *Bridge*-patternet som et abstrakt lag over klassen.

I tillegg vil det ikke være mulig å operere med metodenavnet `drawClock` i klassene for klokkekomponentene, siden disse arver fra andre klasser hvor metodenavnet `operation` er benyttet. Dette navnet må brukes videre i subklassene.

Når man skal implementere hele eksempelet ved hjelp av det generiske idiom, vil man også selvfølgelig være nødt til å legge inn eksplisitte kastinger alle steder hvor man benytter det abstrakte biblioteket.

5.6 Sammendrag

Jeg har i dette kapittelet presentert tre ulike design-patterns og et eksempel hvor man integrerer alle disse i samme program. Ved å gjøre dette, har jeg blitt kjent med design-patterns som hjelp til å finne gode løsninger i bruker-programmer, samtidig som jeg har fått testet ut hvor hensiktsmessig det er med utvidelser av generiske mekanismer i et programmeringsspråk.

På den ene siden har jeg erfart at design-patterns ofte omhandler vel så mye strukturering av klasser og objekter som de inneholder konkret kode. Med andre ord blir det ikke så mye ”kjøtt igjen på bena” når design-patternene er abstrahert bort i fra en konkret kontekst.

Likevel mener jeg at nytten av å ha implementert visse design-patterns i et bibliotek bør vurderes, slik at man i et program automatisk kan hente inn en struktur som kan være hensiktmessig å bruke. På denne måten blir man ”tvunget” til å benytte design-patternene på ”riktig” måte. Fleksibel bruk av disse synes imidlertid å forutsette en eller annen form for multippel arv.

Som jeg viste tidligere i kapittelet, vil noen problemer løses best ved hjelp av generiske klasser, mens andre vil mest naturlig implementeres som generiske pakker. Det er heller ikke noe i veien for at disse mekanismene skal kunne fungere godt sammen, jamfør klassen `JClock`. Jeg mener derfor at Java kan tjene på å bli utvidet med begge disse mekanismene.

Da jeg så på det kombinerte eksempelet som omhandlet klokker, der alle tre patternene inngikk (noen implementert med parametriserte klasser og andre med generiske pakker), var det en viktig observasjon at det fort meldte seg et behov for multippel arv, og at om man hadde full multippel arv, løste det hele seg greit. Statisk multippel arv løste de samme problemene et stykke på vei, men kom til kort i mer kompliserte situasjoner.

En annen erfaring jeg har fått rundt implementasjon i dette kapittelet, er at generiske mekanismer på den ene siden, tilfører mye til et språk, men det vil som regel være mulig, uten så fryktelig mye mer arbeid, å implementere samme kode ved hjelp av det generiske idiom. Selvfølgelig må man da legge inn castinger, noe som ikke er ønskelig.

Multippel arv derimot, er litt annerledes. Hvis man ikke har mulighet for å benytte dette i et språk, er man nødt til å strukturere klasser annerledes for å få gjort det samme. I dette eksempelet var dette tilfellet med klassen `JClock`.

6 Oppsummering

Denne hovedoppgaven har hatt som mål å vurdere eventuelle utvidelser av Java med generiske typer. Ønsket som ligger til grunn for dette, er å øke muligheten for å kunne utvikle gjenbrukbar kode som:

- er naturlig å skrive som separate klasser eller pakker.
- vil gli naturlig inn i de programmer der den brukes; både typemessig og på annen måte.
- vil være like effektiv som om den ikke var programmert separat.

Dette arbeidet startet med å gjennomgå forskjellige generiske mekanismer som benyttes i ulike språk, og vurdere fordeler og ulemper ved disse. Jeg har blant annet sett på templatesmekanismen som benyttes i C++, generiske pakker fra Ada og også noe på generiske typer i Eiffel. Under dette punktet har jeg også vurdert styrken til objektorienteringen som sådan, og sett på hvor kraftig programmering ved hjelp av det generiske idiom kan være.

Ut i fra disse erfaringene har jeg gruppert de ulike generiske mekanismene i tre grupper, med økende grad av språklig integrasjon av mekanismen:

1. En ren makrobasert mekanisme, gjerne implementert ved hjelp av en preprosessor, hvor hver ny instansiering resulterer i ny programkode.
2. De generiske typene er en del av språket og har dermed en strikt syntaks. De generiske parameterne er imidlertid ikke fullstendig spesifisert, slik at det ikke vil være mulig å utføre en fullstendig semantisk sjekk av koden.
3. I den mest integrerte varianten har de generiske typene full spesifisering, og det vil være mulig å utføre en fullstendig semantisk sjekk av en pakke selv uten at man kjenner den faktiske bruken av den.

I resten av fremstillingen har jeg konsentrert meg om mekanismer tilhørende den tredje gruppen. Disse er spesielt interessante ved at de blant annet tillater full typesjekking av generiske pakker, og hvis mekanismen i tillegg er homogent implementert, vil det også være mulig å generere kode før man kjenner bruken av pakka.

Innføring av generiske typer i Java er en pågående debatt i litteratur og på konferanser, og jeg har herfra sett på fire ulike forslag som alle inngår i den tredje gruppen. Jeg har vurdert disse opp mot hverandre, samtidig som jeg har testet dem ut på ulike generiske konstruksjoner av varierende kompleksitet.

Ut i fra disse sammenligningene, valgte jeg å ta utgangspunkt i ett av forslagene, foreslått av Myers mfl. [6].

For å separat kunne typesjekke generiske klasser (i gruppe 3 over), er man nødt til å kunne beskranke parameterne som sendes inn. De foreslåtte mekanismene løser dette noe forskjellig, men hovedforskjellen er at noen benytter nøkkelordene *implements* og *extends*, mens andre benytter *where-clauses*.

Mekanismen til Myers [6] benytter i utgangspunktet *where-clauses*, men ved å kombinere dette med bruk av *implements* og *extends*, blir mekanismen mer fleksibel. Selv om slike utvidelser gjør mekanismen noe mer komplisert, mener jeg dette er en rimelig pris å betale siden de ulike måtene å beskranke på, har styrker på hver sine områder, og begge er relativt viktige.

De fleste av de foreslåtte mekanismene fra kapittel 3 har mange likhetstrekk, og de omhandler stort sett generiske typer på klassenivå. Siden biblioteker i Java som regel samles i pakker, kunne det kanskje være en fordel og også tilby generiske typer på pakkenivå. Jeg har derfor også sett på ideer i denne retning som i sin tid var foreslått som utvidelser av språket Simula. Disse ideene har jeg modifisert noe slik at de skal passe til Java, og deretter forsøkt dem på eksempler og vurdert dem opp mot de mer tradisjonelle generiske typene på klassenivå.

Ut i fra disse sammenligningene fant jeg at mekanismene hadde styrker på hver sine områder. Parametriserte klasser synes best når man vil arbeide med enkeltstående klasser, samtidig som man da lettere kan benytte dem sammen med andre klasser man selv ikke har definert. Dette er blant annet tilfellet i forskjellige ”mengde”-klasser som lister, stakker osv.

Generiske pakker derimot vil oftest fungere best når man har biblioteker bestående av flere klasser med gjensidige referanser, og hvor alle skal subklasses ved et konkret brukstilfelle. Eksempelet rundt grafer illustrerte dette; i tillegg til at noen design-pattern også viste seg best implementert ved generiske pakker. Flere design-pattern har i utgangspunktet en struktur som passer til generiske pakker siden de er satt sammen av flere klasser som skal spille sammen.

Som jeg har nevnt tidligere i oppgaven, vil det alltid være en avveining mellom ønsket om mer funksjonalitet og å holde et språk så enkelt som mulig. I denne oppgaven har jeg presentert eksempler som viser at det kan være nyttig med generiske mekanismer både på klasse- og pakkenivå, og det ser også ut som at de presenterte mekanismene kan leve side ved side uten at de ødelegger for hverandre.

Jeg mener derfor man bør vurdere om ikke en utvidelse av Java, skal omfatte begge disse forslagene. Det kunne kanskje også være aktuelt å kombinere disse teknikkene slik at man i tillegg innfører vanlige typeparametere på de generiske pakkene, men dette har jeg ikke gått i detalj på grunnet omfanget av oppgaven. I et eventuelt videre arbeid, kunne man sett nærmere på hvor nyttig en slik utvidelse ville vært, og hvor mye mer komplisert mekanismen da ville blitt.

Siden denne oppgaven i hovedsak skal vurdere generiske typer, har jeg kun raskt presentert forskjellige sider rundt multippel arv. Jeg har sett på to forskjellige former for multippel arv. Den ene er ”full multippel arv” omtrent som i C++, mens den andre er en enklere form, kalt ”statisk multippel arv”, som er knyttet til forslaget om generiske pakker.

Til sist i oppgaven har jeg tatt for meg tre forskjellige design-patterns og sett på hvordan de kunne vært implementert som bibliotekskklasser, og hvilke mekanismer det vil være hensiktsmessig å bruke under implementasjonen. Under dette forsøket, har jeg samtidig sett på hva de nye mekanismene har tilført Java, og hvordan design-patternene kunne blitt implementert hvis man måtte programmere ved hjelp av det generiske idiom, slik man må i Java i dag.

En observasjon i denne forbindelse, var at design-patternene egentlig består av ganske lite kode når de blir abstrahert bort i fra alle kontekster. For flere av design-patternene står man stort sett igjen med en klassestruktur bestående av relativt tomme klasser med abstrakte navn. Noen inneholder også en del kode, men ikke fullt så mye som jeg hadde forestilt meg på forhånd. Likevel fikk jeg testet ut de nye mekanismene ganske godt da typeproblematikken i stor grad er knyttet til klassestrukturen som sådan.

Alt i alt har jeg altså kommet fram til at følgende utvidelser av nåværende Java bør vurderes:

- Som et ledd i å kunne lage mer fleksible klasser som skal kunne brukes i forskjellige sammenhenger, har jeg sett på en mekanisme som introduserer generiske parametere på klassenivå.
- Som et ledd i å kunne lage mer fleksible biblioteker, har jeg sett på en mekanisme som gjør det mulig å innføre generiske klasser/parametere på pakkenivå.

En eventuelt innføring av disse mekanismene, vil gjøre det enklere å skrive mer generelle biblioteker som kan benyttes i ulike sammenhenger. For fleksibelt å kunne kombinere disse, vil det raskt melde seg et behov for en eller annen form for multippel arv, men man bør nok grundigere vurdere denne delen for å avgjøre hvilken form som vil egne seg best.

Til sammen vil disse generiske utvidelsene være en forbedring i forhold til de eksisterende mekanismer i Java. Teknikkene har bedre uttrykkskraft, og gjør dermed at språket blir kraftigere enn det er.

TILLEGG

A Klokkeeksempel

I kapittel 5.5 jobbet jeg med et eksempel hvor jeg tok utgangspunkt i `swing`-biblioteket som følger med Java 1.2. Jeg lagde en ny klasse `JClock` som kunne brukes på tilsvarende måte som de andre klassene i biblioteket. I denne utviklingen benyttet jeg de tre design-patternene som jeg tidligere i kapittelet hadde gjennomgått.

Da jeg presenterte eksempelet, tok jeg utgangspunkt i at jeg hadde tilgjengelig de nye mekanismene jeg har kommet fram til i løpet av oppgaven. Jeg diskuterte også litt om hvordan eksempelet kunne implementeres i Java 1.2, hvor man i stedet blant annet må ta utgangspunkt i det generiske idiom.

I dette appendikset står hele koden for eksempelet, men det er som sagt, løst noe annerledes da jeg blant annet ikke kan bruke multippel arv. I koden har jeg heller ikke plassert de forskjellige pattern-klassene i biblioteker, men brukt dem direkte siden alle klassene lå på samme sted. Likevel vil ikke dette ha noe å si for implementasjonen.

A.1 Program.java

Dette er selve hovedprogrammet i eksempelet. Det er med andre ord denne klassen som starter det hele opp. I det opprinnelige eksempelet hadde klassen direkte tilknytninger til objekter av klassen `JClock`, men siden jeg her ikke kan benytte multippel arv og siden det krever litt kode å sette opp vinduer, har jeg opprettet en mellomklasse `ClockWindow` som inneholder noe av funksjonaliteten.

Klassen `Program` definerer først at programmet skal ha at Windows-utseende, deretter oppretter den objektet som holder på klokkeslettet, før den oppretter to uavhengige vinduer med hver sin klokke. Klokkene får tilordnet det felles tidsobjektet, og til slutt startes selve klokkesystemet.

```
1     import java.io.*;
2     import javax.swing.*;
3
4
5     public class Program {
6
7         public static void main(String[] args) {
8             try {
9                 UIManager.setLookAndFeel(
10                "com.sun.java.swing.plaf.metal.MetalLookAndFeel");
11            }
12            catch (Exception e) {
13            }
14
15            /* The timer goes in it's own thread. */
16            SystemTime timer = new SystemTime();
```

```
17            Thread timerThread = new Thread(timer);
18            ClockWindow winClock1 = new ClockWindow();
19            ClockWindow winClock2 = new ClockWindow();
20
21            /* Sets the timer as subject for the clock observers. */
22            winClock1.setSubject(timer);
23            winClock2.setSubject(timer);
24            /* Sets the clocks as observers for the timer. */
25            timer.attach(winClock1);
26            timer.attach(winClock2);
27            /* The timer starts and updates the clocks. */
28            timerThread.start();
29        }
30
31    }
```

A.2 ClockWindow.java

Denne klassen inneholder noe av funksjonaliteten som opprinnelig var tenkt plassert i `JClock`, men uten multippel arv må noe løses annerledes. Klassen inneholder kode for å opprette et vindu, samtidig som den linker inn et objekt av typen `JClock`, som er en ekte subklasse til `JComponent`. Klassen `ClockWindow` er en subklasse av `VisualClock`, og den inneholder dermed en metode `update`, som blir kalt når klokkeslettet er endret. Metoden `update` sender bare et kall videre til `JClock` om at den må oppdatere seg.

```
32    import java.awt.*;
33    import java.awt.event.*;
34    import javax.swing.*;
35
36
37    public class ClockWindow extends VisualClock implements ActionListener {
38        JFrame frame;
39        JClock clock;
40
41
42        public ClockWindow() {
43            frame = new JFrame("Clock");
44            frame.setSize(140, 170);
45            /* Puts the menu into the frame window. */
46            insertMenu();
47
48            frame.addWindowListener(new WindowAdapter() {
49                public void windowClosing(WindowEvent w) {
50                    System.exit(0);
51                }
52            });
53
54            /* Insert the clock element. */
55            clock = new JClock(clock.ANALOG);
56            frame.getContentPane().add(clock, BorderLayout.CENTER);
57        }
58
59
60        private void insertMenu() {
61            /* Inserts the menu bar. */
62            JMenuBar menuBar = new JMenuBar();
63            frame.setJMenuBar(menuBar);
64
65            /* Build the pull down menu. */
66            JMenu menu = new JMenu("Clock");
67            menu.setMnemonic(KeyEvent.VK_C);
68            menu.getAccessibleContext().setAccessibleDescription(
69                                        "The only menu in this program");
70            menuBar.add(menu);
71
```

```
72            /* Build the radiobutton menu items. */
73            ButtonGroup group = new ButtonGroup();
74            JRadioButtonMenuItem rbMenuItem;
75
76            rbMenuItem = new JRadioButtonMenuItem("Analog");
77            rbMenuItem.addActionListener(this);
78            rbMenuItem.setSelected(true);
79            rbMenuItem.setMnemonic(KeyEvent.VK_A);
80            group.add(rbMenuItem);
81            menu.add(rbMenuItem);
82
83            rbMenuItem = new JRadioButtonMenuItem("Digital");
84            rbMenuItem.addActionListener(this);
85            rbMenuItem.setMnemonic(KeyEvent.VK_D);
86            group.add(rbMenuItem);
87            menu.add(rbMenuItem);
88
89            /* Inserts the separator. */
90            menu.addSeparator();
91
92            /* Build the last menu item. */
93            JMenuItem menuItem = new JMenuItem("Quit");
94            menuItem.addActionListener(this);
95            menuItem.setMnemonic(KeyEvent.VK_Q);
96            menu.add(menuItem);
97        }
98
99
100       /* This is the menu listener. */
101       public void actionPerformed(ActionEvent e) {
102           if (e.getActionCommand().equals("Digital"))
103               clock.setClockType(clock.DIGITAL);
104           else if (e.getActionCommand().equals("Analog"))
105               clock.setClockType(clock.ANALOG);
106           else if (e.getActionCommand().equals("Quit"))
107               System.exit(0);
108       }
109
110
111       /* Overrides the method in Clock to let the clock print it's */
112       /* new time.                                                 */
113       public void update() {
114           /* Updates the clock. */
115           super.update();
116           /* Prints the clock. */
117           drawClock();
118       }
119
120
121       public void drawClock() {
122           clock.setTime(hour, minute, second);
123           frame.setVisible(true);
124       }
125
126   }
```

B *Observer*-pattern

Dette patternet blir ganske likt det som står i det opprinnelige eksempelet, men grunnet problemer rundt multippel arv, har jeg valgt å ha med klassen `VisualClock`, i stedet for å la `JClock` være en direkte subklasse av `Observer`.

B.1 Observer.java

Klassen er slik den er presentert tidligere i oppgaven, og den er definert abstrakt slik at man må subklasse den ved bruk. Det skal med andre ord ikke opprettes noen instanser av denne klassen.

```
127   public abstract class Observer {
128       private Subject subject;
129
130       public void setSubject(Subject subj) {
131           subject = subj;
132       }
133
134       public Subject subject() {
135           return subject;
136       }
137
138       /** This method is called by the subject.
139           Must be re-declared. */
140       public abstract void update();
141   }
```

B.2 Subject.java

Klassen er, på samme måte som `Observer`, lik den som er presentert tidligere i oppgaven, og den er definert abstrakt slik at man må subklasse den ved bruk. Det skal heller ikke opprettes noen instanser av denne klassen.

```
142   import java.util.*;
143
144
145   public abstract class Subject {
146       private Vector observers = new Vector();
147
148
149       /** Adds the observer if it doesn't exist. */
150       public void attach(Observer obs) {
151           if (!observers.contains(obs))
152               observers.add(obs);
153       }
154
155
156       /** Removes the observer if it exists. */
157       public void detach(Observer obs) {
158           observers.remove(obs);
159       }
160
```

```
161
162       /** Runs the update-method on every observer. */
163       public void notifyObservers() {
164           for (int i = 0; i < observers.size(); i++)
165               ((Observer)observers.elementAt(i)).update();
166       }
167
168   }
```

B.3 VisualClock.java

I klokkeeksempelet er `JClock` i utgangspunktet en subklasse til `Observer`, men i dette eksempelet har jeg puttet inn `VisualClock` som et mellomledd da `JClock` ikke kan ha flere superklasser.

Tidligere i oppgaven brukes det en egen klasse `Time` for å holde på klokkeslettet. Her har jeg valgt å splitte dette opp i timer, minutter og sekunder.

```
169   public class VisualClock implements Observer {
170       protected int hour, minute, second;
171
172
173       /** This method is run by the timer subject, and tells the
174           clock to update it's time. */
175       public void update() {
176           hour = ((SystemTime)subject).hours();
177           minute = ((SystemTime)subject).minutes();
178           second = ((SystemTime)subject).seconds();
179       }
180
181   }
```

B.4 SystemTime.java

Denne klassen er lik den som opprinnelig ble presentert. Klassen går i en egen tråd, og gir hvert sekund beskjed til objektene som har meldt sin interesse om at klokkeslettet er endret.

Tidligere i oppgaven brukes det en egen klasse `Time` for å holde på klokkeslettet. Her har jeg valgt å splitte dette opp i timer, minutter og sekunder.

```
182   import java.util.*;
183
184
185   /** Each timer is in it's own thread. */
186   public class SystemTime extends Subject implements Runnable {
187       private Calendar time;
188       /* Uses this as signal to stop the thread. */
189       private boolean finished;
190
191
192       public void run() {
193           while (!finished) {
194               /* Gets the time from the system. */
195               time = Calendar.getInstance();
196               /* Notifies the observers to tell them to */
197               /* draw themself. */
198               notifyObservers();
199               /* Waits one second before it updates itself again. */
200               try {
201                   Thread.sleep(1000);
202               }
203               catch (Exception e) {
```

```
204                 }
205             }
206         }
207
208
209         public void setFinished(boolean value) {
210             finished = value;
211         }
212
213
214         public int hours() {
215             return time.get(Calendar.HOUR_OF_DAY);
216         }
217
218
219         public int minutes() {
220             return time.get(Calendar.MINUTE);
221         }
222
223
224         public int seconds() {
225             return time.get(Calendar.SECOND);
226         }
227
228     }
```

C *Composite*-pattern

Klassene som er omtalt her, er de som utgjør *Composite*-patternet. I tillegg beskrives klassene som utgjør elementene i klokka, som visere, urskive osv.

C.1 Component.java

Dette er den ytterste superklassen i *Composite*-patternet, og den er lik som i eksempelet. Siden den generelle metoden `operation` skal kunne ta forskjellig type parametere, er parameteren typet med klassen `Object`.

```
229   import java.awt.*;
230
231
232   public abstract class Component {
233       /** This method is called for all the components. */
234       public abstract void operation(Object param);
235       public abstract void add(Component comp);
236       public abstract void remove(Component comp);
237   }
```

C.2 Composite.java

Klassen er som i hovedeksempelet og inneholder kode for å videresende `operation` til hver av primitivene og undergruppene som `Composite`-objektet inneholder.

```
238   import java.util.*;
239   import java.awt.*;
240
241
242   public abstract class Composite extends Component {
243       private Vector children = new Vector();
244
245
246       /** Adds the component if it doesn't exist. */
247       public void add(Component comp) {
248           if (!children.contains(comp))
249               children.add(comp);
250       }
251
252
253       /** Removes the component if it exists. */
254       public void remove(Component comp) {
255           children.remove(comp);
256       }
257
258
259       /** Runs the operation-method on every child. */
260       public void operation(Object param) {
261           for (int i = 0; i < children.size(); i++)
262               ((Component)children.elementAt(i)).operation(param);
263       }
264
265   }
```

C.3 Leaf.java

Klassen `Leaf` er som tidigere beskrevet og inneholder ikke noe særlig kode. Den inneholder de to grupperingsmetodene `add` og `remove`, og disse "termineres" samtidig som de ikke gjør noe som helst. Med andre ord kan de ikke redefineres i subklasser.

```
266   public abstract class Leaf extends Component {
267
268       public final void add(Component comp) {
269       }
270
271       public final void remove(Component comp) {
272       }
273
274   }
```

C.4 JAnalogClockImp.java

Dette er klassen som representerer hele den analoge versjonen av klokka. Siden klassen samtidig skal fungere i *Bridge*-patternet, har den et navn som tilsvarer de andre klassene som er definert der.

I utgangspunktet skal ikke klassen inneholde så mye kode, bortsett fra at underkomponentene må legges inn (noe som gjøres i konstruktøren). Siden det ikke er mulighet for navneendringer, inneholder klassen en metode med navn `drawClock`. Det eneste denne gjør, er å kalle metoden `operation` som sørger for opptegning av underkomponentene.

```
275   import java.awt.*;
276
277
278   public class JAnalogClockImp extends Composite implements JClockImp {
279
280       public JAnalogClockImp() {
281           /* When the analog clock is made, all it's components are */
282           /* created at the same time so the clock can be painted.  */
283           add(new ClockBackground());
284           add(new ClockHands());
285       }
286
287
288       public void drawClock(Graphics g, int hour, int min, int sec,
289                             int width, int height) {
290           /* Call operation to print the whole clock using all the */
291           /* clock components.                                     */
292           operation(new ClockInfo(g, width, height, hour, min, sec));
293       }
294
295   }
```

C.5 ClockHands.java

Dette er klassen som "holder" på alle viserne. Bortsett fra å opprette objekter for viserne, inneholder ikke klassen noe mer funksjonalitet. Resten arves fra superklassen `Composite`.

```
296   import java.awt.*;
297
```

```
298
299   public class ClockHands extends Composite {
300
301       public ClockHands() {
302           super();
303           add(new ClockHourHand());
304           add(new ClockMinuteHand());
305           add(new ClockSecondHand());
306       }
307
308   }
```

C.6 ClockHourHand.java

Klassen inneholder funksjonalitet for å tegne opp seg selv innenfor det området klokka skal dekke.

```
309   import java.awt.*;
310
311
312   public class ClockHourHand extends Leaf {
313
314       public void operation(Object param) {
315           ClockInfo p = (ClockInfo)param;
316
317           int xCen = p.width / 2;
318           int yCen = p.height / 2;
319
320           int xh = (int)(Math.cos((p.hour * 30 + p.min / 2) *
321                                   3.14f / 180 - 3.14f / 2) *
322                                   (xCen * 0.5) + xCen);
323           int yh = (int)(Math.sin((p.hour * 30 + p.min / 2) *
324                                   3.14f / 180 - 3.14f / 2) *
325                                   (yCen * 0.5) + yCen);
326
327           p.g.drawLine(xCen, yCen - 1, xh, yh);
328           p.g.drawLine(xCen - 1, yCen, xh, yh);
329       }
330
331   }
```

C.7 ClockMinuteHand.java

Klassen inneholder funksjonalitet for å tegne opp seg selv innenfor det området klokka skal dekke.

```
332   import java.awt.*;
333
334
335   public class ClockMinuteHand extends Leaf {
336
337       public void operation(Object param) {
338           ClockInfo p = (ClockInfo)param;
339
340           int xCen = p.width / 2;
341           int yCen = p.height / 2;
342
343           int xm = (int)(Math.cos(p.min * 3.14f / 30 - 3.14f / 2) *
344                                   (xCen * 0.9) + xCen);
345           int ym = (int)(Math.sin(p.min * 3.14f / 30 - 3.14f / 2) *
346                                   (yCen * 0.9) + yCen);
347
348           p.g.drawLine(xCen, yCen - 1, xm, ym);
349           p.g.drawLine(xCen - 1, yCen, xm, ym);
350       }
351
```

```
352   }
```

C.8 ClockSecondHand.java

Klassen inneholder funksjonalitet for å tegne opp seg selv innenfor det området klokka skal dekke.

```
353   import java.awt.*;
354
355
356   public class ClockSecondHand extends Leaf {
357
358       public void operation(Object param) {
359           ClockInfo p = (ClockInfo)param;
360
361           int xCen = p.width / 2;
362           int yCen = p.height / 2;
363
364           int xs = (int)(Math.cos(p.sec * 3.14f / 30 - 3.14f / 2) *
365                                   (xCen * 0.9) + xCen);
366           int ys = (int)(Math.sin(p.sec * 3.14f / 30 - 3.14f / 2) *
367                                   (yCen * 0.9) + yCen);
368
369           p.g.drawLine(xCen, yCen, xs, ys);
370       }
371
372   }
```

C.9 ClockBackGround.java

Klassen inneholder funksjonalitet for å tegne opp seg selv innenfor det området klokka skal dekke.

```
373   import java.awt.*;
374
375
376   public class ClockBackground extends Leaf {
377
378       public void operation(Object param) {
379           ClockInfo p = (ClockInfo)param;
380
381           int xCen = p.width / 2;
382           int yCen = p.height / 2;
383
384           p.g.drawOval(0, 0, p.width, p.height);
385           p.g.drawString("9", 4, yCen + 4);
386           p.g.drawString("3", p.width - 10, yCen + 4);
387           p.g.drawString("12", xCen - 6, 15);
388           p.g.drawString("6", xCen - 3, p.height - 5);
389       }
390
391   }
```

D *Bridge*-pattern

Siden Java ikke støtter multippel arv, er det umulig å trekke *Bridge*-patternet ut som egne klasser siden `JClock` i tillegg må være en ekte subklasse av `JComponent`. Jeg har derfor flettet *Bridge*-patternet inn i klassen, slik at `JClock` derfor inneholder `imp`-pekeren, og denne er typet med interfacet `JClockImp`.

Klassen `JAnalogClockImp` er definert tidligere under *Composite*-patternet.

D.1 JClock.java

Klassen `JClock` representerer selve den nye `swing`-komponenten, og den inneholder blant annet metoden `paintComponent` som automatisk blir kalt når vinduet trenger å tegnes opp på nytt. Denne oppdateringen "tvinges" fram i metoden `setTime`, som kalles når klokkeslettet oppdateres.

```
392   import java.awt.*;
393   import javax.swing.*;
394
395
396   public class JClock extends JComponent {
397       public final static int ANALOG = 1;
398       public final static int DIGITAL = 2;
399
400       private int hour, min, sec;
401       private JClockImp clockImp;
402
403
404       public JClock(int type) {
405           /* You choose a clock type when you make it. */
406           setClockType(type);
407       }
408
409
410       public void setTime(int aHour, int aMin, int aSec) {
411           hour = aHour;
412           min = aMin;
413           sec = aSec;
414           /* Signal that the time has changed so the clock must be */
415           /* painted again.                                        */
416           repaint(new Rectangle(getWidth(), getHeight()));
417       }
418
419
420       /* This method is called automatic since this is a subclass */
421       /* of JComponent.                                           */
422       public void paintComponent(Graphics g) {
423           /* Paints the background. */
424           super.paintComponent(g);
425           /* Draws the clock. */
426           clockImp.drawClock(g, hour,  min, sec,
427                              getWidth() - 1, getHeight() - 1);
428       }
429
430
```

```
431       public void setClockType(int type) {
432           if (type == ANALOG)
433               clockImp = new JAnalogClockImp();
434           else
435               clockImp = new JDigitalClockImp();
436       }
437   }
```

D.2 JClockImp.java

Dette er interfacet som definerer grensesnittet for å tegne opp en klokke, og det implementeres av både klassen som representerer analoge og digitale klokker.

```
438   import java.awt.*;
439
440
441   public interface JClockImp {
442       public void drawClock(Graphics g, int hour, int min, int sec,
443                             int width, int height);
444   }
```

D.3 JDigitalClockImp.java

Denne klassen inneholder kode for å tegne opp den digitale klokka. For at klokka skal fylle ut den plassen den er blitt tilegnet, regnes det ut hvilken skriftstørrelse som må brukes. Metoden for å regne ut skriftstørrelsen, er basert på en lignende metode som står i boka *"The JFC Swing Tutorial, A Guide to Constructing GUIs"* [14].

```
445   import java.awt.*;
446
447
448   public class JDigitalClockImp implements JClockImp {
449       private Font font = null;
450       private FontMetrics fontMetrics = null;
451       private int oldWidth, oldHeight;
452
453
454       public void drawClock(Graphics g, int hour, int min, int sec,
455                             int width, int height) {
456           /* Makes the string to draw. */
457           String time = ((hour < 10) ? "0" + hour : "" + hour) + ":";
458           time += ((min < 10) ? "0" + min : "" + min) + ":";
459           time += (sec < 10) ? "0" + sec : "" + sec;
460
461           /* Calculates a properly font size if it is necessary. */
462           if (font == null || oldWidth != width || oldHeight != height) {
463               pickFont(g, time, width, height);
464               /* Saves the last window size. */
465               oldWidth = width;
466               oldHeight = height;
467           }
468           /* Draws the string. */
469           g.setFont(font);
470           g.drawString(time, 0, fontMetrics.getAscent());
471       }
472
473
474       /* This method calculates what font to use to let the text */
475       /* fit into the rectangle.                                 */
476       private void pickFont(Graphics g, String string,
477                             int xSpace, int ySpace) {
478           final Font biggestFont = new Font("SansSerif", Font.PLAIN, 256);
```

```
479           final int minFontSize = 6;
480
481           boolean fontFits = false;
482           Font currentFont = biggestFont;
483           FontMetrics currentMetrics = g.getFontMetrics(currentFont);
484           int size = currentFont.getSize();
485           String name = currentFont.getName();
486           int style = currentFont.getStyle();
487
488           while (!fontFits) {
489               if ((currentMetrics.getHeight() <= ySpace) &&
490                   (currentMetrics.stringWidth(string) <= xSpace)) {
491                   fontFits = true;
492               }
493               else {
494                   if (size <= minFontSize)
495                       fontFits = true;
496                   else {
497                       currentFont = new Font(name, style, size -= 2);
498                       currentMetrics = g.getFontMetrics(currentFont);
499                   }
500               }
501           }
502           fontMetrics = currentMetrics;
503           font = currentFont;
504       }
505
506   }
```

Referanser

[1] Jeff Langr. *Essential Java Style – Patterns for Implementation*. Prentice Hall PTR, 2000, ISBN 0-13-085086-1.

[2] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design-patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley Corporate Professional, 1994, ISBN 0-201-63361-2.

[3] Mark Grand. *Patterns in Java, volume 1*. Wiley, 1998, ISBN 0-471-25839-3.

[4] Simon Johnston, Addison Wesley Longman. *Ada-95 for C and C++ Programmers*.
Tilgjengelig på: http://www.ankh-morpork.com/CppToAda/.

[5] Narain Gehani. *UNIX Ada programming*. AT&T Bell Telephone Laboratories, Incorporated, 1987, ISBN 0-13-938325-5.

[6] Gilad Bracha, Martin Odersky, David Stoutamire og Philip Wadler. *Making the future safe for the past: Adding Genericity to the Java Programming Language*. Conferense Proceeding of OOPSLA '98. Sigplan, Volum 33.

[7] Bjarne Stroustrup. *The design and evolution of C++*, kapittel 15: *Templates*, 1994, ISBN 0-201-54330-3.

[8] Ole Agesen, Stephen N. Freund og John C. Mitchell. *Adding Type Parameterization to the Java Language*. Conference on Object-Oriented Programming, Systems, Languages and Applications, side 215-230, 1997.

[9] Stein Krogdahl, Arne Wang, Peter Jensen og Jo Piene. *A Module-mechanism for Flexible Code Reuse*. Juni 1993.

[10] Stein Krogdahl. *On Modernizing the Simula Language*. November 15, 1986. Technical Report, institutt for informatikk, universitetet i Oslo, 1986.

[11] Kresten Krab Thorup. *Genericity in Java with Virtual Types*. Springer, Proceedings ECOOP '97 Object-Oriented Programming. Lecture Notes in Computer Science 1241.

[12] Bjarne Stroustrup. *The design and evolution of C++*, kapittel 12: *Multiple Inheritance*, 1994, ISBN 0-201-54330-3.

[13] Bjarne Stroustrup. *Multiple Inheritance for C++*. Proc. EUUG Spring Conference, May 1987. Also, USENIX Computer Systems, Vol 2 No 4. Fall 1989.

[14] Walrath Campione. *The JFC Swing Tutorial, A Guide to Constructing GUIs*. Addison Wesley, 1999, ISBN 0-201-43321-4.

[15] Andrew C. Myers, Joseph A. Bank og Barbara Liskov. *Parameterized Types for Java*. Proceedings of the 24th ACM Symposium on Principles of Programming Languages, Paris, France, January 1997.

[16] OMG Unified Modeling Language, Specifications, versjon 1.3, June 1999.

Tilgjenglig på: http://www.omg.org/.

[17] Robert Cartwright og Guy L. Steele Jr. *Compatible Genericity with Run-time Types for the Java Programming Language*. Conferense Proceeding of OOPSLA '98. Sigplan, Volum 33.

[18] Martin Odersky and Philip Wadler. *Pizza into Java: Translating theory into practice*. Symbosium on Principles of Programming Languages, side 146-159, ACM, 1997.

[19] Mark Day, Robert Gruber, Barbara Liskov og Andrew C. Myers. *Subtypes vs. where clauses: Constraining parametric polymorphism*. In Proc. of OOPSLA '95, side 156-158, Austin TX, oktober 1995. ACM SIGPLAN Notices 30(10).

[20] Barbara Liskov, Dorothy Curtis, Mark Day, Sanjay Ghemawat, Robert Gruber, Paul Johnson og Andrew C. Myers. *Theta Reference Manual*. Programming Methodology Group Memo 88, MIT Laboratory for Computer Science, Cambridge, MA, februar 1994.

Tilgjengelig på: http://www.pmg.lcs.mit.edu/papers/thetaref/.

[21] Tower Technology Corporation, 1996. *Getting Started with TowerEiffel, Genericity*.

Tilgjengelig på: http://www.cs.herts.ac.uk/tower/tut.htm.

[22] Kresten Krab Thorup og Mads Torgersen. *Unifying Genericity*. ECOOP99, Springer, 1999.

[23] Bjørn Willassen. *Generiske mekanismer i BETA*. Hovedfagsoppgave, institutt for informatikk, universitetet i Oslo, høst 2000.

[24] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King og Shlomo Angel. *A Pattern Language*. Oxford University Press, New York, 1977.

[25] P. Canning, W. Cook, W. Hill, J. Mitchell og W. Olthoff. *F-Bounded Polymorphism for Object-Oriented Programming*, hentet fra *Functional Prog. and Computer Architecture*, side 273-280, 1989.

[26] Barbara Liskov et al. *CLU Reference Manual*. Springer LNCS 114, Berlin, 1981.

[27] Lars Tafjord og Holm-Kjetil Holmsen. *SiMod, Et Simula-basert språk med moduler og multippel arv*. Hovedfagsoppgave, institutt for informatikk, universitetet i Oslo, 1991.

[28] Ellen Agerbo og Aino Cornils. *Theory of Language Support for Design Patterns*. Master's Thesis, Department of Computer Science, University of Aarhus, Denmark, 1997.

[29] Barbara Liskov, Alan Snyder, Russell Atkinson og Craig Schaffert. *Abstraction mechanisms in CLU*. Communications of the ACM, 20(8):564-576, august 1977.